Department of Computing Science,
University of Newcastle upon Tyne

Asynchronous Circuit Synthesis by Direct Mapping: Interfacing to Environment

A. Bystrov and A. Yakovlev

Technical Report Series

October, 2001

Contact:
A.Bystrov@ncl.ac.uk

Alex.Yakovlev@ncl.ac.uk

Supported by EPSRC grant GR/M94366

Published by the University of Newcastle upon Tyne,
Department of Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, UK

Asynchronous Circuit Synthesis by Direct Mapping: Interfacing to Environment

A. Bystrov and A. Yakovlev

12 October 2001

Abstract

Direct mapping helps avoid algorithmic complexity which is inherent in logic synthesis methods. However, existing techniques for direct mapping of Petri net specifications to asynchronous control circuit do not deliver in performance due to logic overhead and inefficient interface to the environment. The paper presents a direct mapping method for Signal Transition Graphs (STGs) targetted at lower latency between input and output events. It is based on two behaviour-preserving transformations applied to the initial STG model: *output exposition* and *environment tracking*. The former allows interface signals to be generated concurrently to internal transitions. The latter prevents creation of coding conflicts. Subsequent refinement combines the use of the tracking and input signals in the control of the output flip-flops so as to optimise the circuit size by removing some tracking components. The depth of final logic in the design examples is one or two gates. The comparison to logic synthesis methods indicates lower output latency and modest size increase. The proposed direct-mapping method allows using fast transistor-level implementations for tracking and output signals with well-localised relative timing constraints.

1 Introduction

Power consumption, electromagnetic compatibility, clock skew, robustness and scalability become increasingly critical issues in modern VLSI circuits. This draws more interest in asynchronous designs as they compare favourably to synchronous circuits with respect to the above criteria. Two main approaches exist in asynchronous circuit design: direct syntax mapping [7, 11, 2, 14, 25] and logic synthesis [18, 5, 9]. In this paper, we focus on the direct mapping of concurrent specifications, defined as Petri nets (PN), into control circuits built of standard logic cells.

The direct mapping approach has a long history originating from Huffman's work [12], where a method of *the one-relay-per-row* realisation of an asynchronous sequential circuit was proposed. This approach was further developed by Unger in [24] and led to the *1-hot state assignment* of Hollaar [11], where a method of concurrent circuit synthesis was described. Hollaar's circuits have inputs that are logic values (signal levels as opposed to signal transitions), which is good for low-level interfacing. They 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 Hollaar's 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). This issue has not been addressed in [11].

The approach of [14] is based on 'distributors' and also uses the 1-hot state assignment, though a different implementation of local states. It uses the so-called *David cells* (DC), which resemble modules described in [6]. As a result, these circuits are *speed independent* (SI) [17] 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 his method to high-level design. An attempt to apply this method at a low-level, where inputs and outputs are signal events of positive or negative polarity, was done in [25], where DC structures controlled output flip-flops. For this, a circuit converting a handshake event into the logic level was designed. Inputs, were however still represented as abstract processes.

As follows from the above review, the subject of interfacing the environment in direct syntax mapping methods was insufficiently studied. This, probably, can be explained by the area of intended application for these methods, which is the design of high-level interfaces, where processes are represented as handshakes on dedicated signals. When applied to low-level specifications, such as STGs, where an event was associated with a single signal transition, these methods were inefficient and would produce large and slow solutions. For example, replacing every signal event in the protocol of a simple 'toggle' element $x+ \rightarrow y_1+ \rightarrow x- \rightarrow y_2+ \rightarrow x+ \rightarrow y_1- \rightarrow x- \rightarrow y_2-$ by a handshake $Req_i+ \rightarrow Ack_i+ \rightarrow Req_i- \rightarrow Ack_i-$ where all actions are performed sequentially, results in a very slow circuit.

The above problem does not exist in logic synthesis methods, aimed at deriving a logic function for every output signal. The inputs of logic functions are the values of input and internal signals, as opposed to signal transitions in direct mapping methods. This explains why logic synthesis tools, such as Petrify [5], are still considered as 'surgical tools', suitable for the design of low granularity asynchronous circuits. However, the recent advance in the PN theory and introduction of the so called 'read-arcs' [16] makes it possible to directly model states of logic signals. In this paper we use this convenient abstraction to represent an asynchronous circuit specification in such a form, which gives a compact and fast implementation by direct mapping.

2 Background

In this section we first introduce a model to specify the behaviour of a synthesised object, subsequently called a *device*, in its interaction with the *environment*. Secondly, a set of modules implementing the elements of such a model by means of *direct syntax mapping* are presented.

2.1 Model

In this paper we use *1-safe* PNs to capture concurrent behaviour of an asynchronous system and *signal transition graphs* (STG) as a circuit specification. Traditionally, a PN (a *net system*) is defined as a tuple $\Sigma = \langle P, T, F, M_0 \rangle$ comprising finite disjoint sets of places P and transitions T, flow relation $F \subseteq (P \times T) \cup (T \times P)$ and initial marking M_0. There is an arc between x and y iff $(x, y) \in F$. The *preset* of a node x is defined as $\bullet x = \{y \mid (y, x) \in F\}$, and the *postset* as $x\bullet = \{y \mid (x, y) \in F\}$. A *marking* is a mapping $M : P \rightarrow N = \{0, 1\}$ (N is binary for 1-safe PNs). It is assumed that $\bullet t \neq \emptyset \neq t\bullet$ for every transition $t \in T$. The evolution of a PN from the initial marking M_0 to a marking M_n by executing transitions results in a *firing sequence* $\sigma = t_1 t_2 \ldots t_n$, where t_i are such that $M_i [t_{i+1}\rangle M_{i+1}$, for $i = 0, \ldots, n-1$. Moreover, M_n is called a *reachable*

marking.

An extension of a PN model is a *contextual net* [16]. It uses additional elements such as non-consuming arcs, which only control enabling of a transition and do not consume tokens. In this paper we use only one type of a non-consuming arc, namely a *read-arc*. A set of read-arcs R can be defined as follows: $R \subseteq (P \times T)$, $R \cap F = \oslash$; there is an arc between x and y iff $(x, y) \in R$.

A Signal Transition Graph (STG) is a PN whose transitions are labelled by signal events, i.e. $STG = \langle P, T, F, R, M_0, \lambda \rangle$, where $\lambda : T \rightarrow A \times \{+, -\}$ is a labelling function and A is a set of signals. Usually, the labelling assigns the same label to several transitions. Note, that a set of read-arcs R has been included into the model of STG, which is an enhancement w.r.t. [20, 4]. Another specific property of an STG is signal consistency, i.e. when the net is executed no signal can have two transitions of the same polarity without having a transition of the opposite polarity between them.

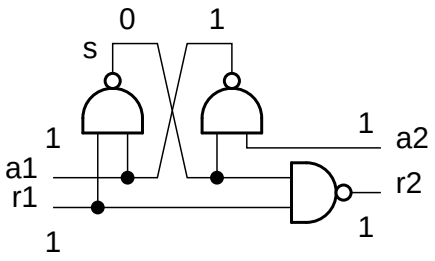
2.2 Direct syntax mapping

In this paper we choose a direct mapping method of [14, 25] as the basis for circuit construction. In this method every place of a PN or an STG is associated with a DC (originates from [6]), whose circuit diagram is shown in Figure 1(a). DCs can be coupled using four-phase handshake interfaces, so that the interface $< r2, a2 >$ of the previous stage DC is connected to the interface $< r1, a1 >$ of the next stage. The shown schematic uses the logic value 0 as an active level in those handshakes (a dual construction is possible with NOR gates and active 1). The operation of a single DC is illustrated by STG in Figure 1(b). Places $p1$ and $p2$ in Figure 1(b) correspond to $r1 = active$ and $r2 = active$. They can be used to model places of a PN as shown in Figure 1(c). The dotted rectangle depicts the transition between $p1$ and $p2$. This transition contains an internal place, where a token 'disappears' for the time $t_{r1+ \rightarrow r2-}$. In most cases this time can be considered as negligible, because it corresponds to a single gate delay. The limitation of the method using DCs is that any loop in the specification PN must contain at least three places [14].

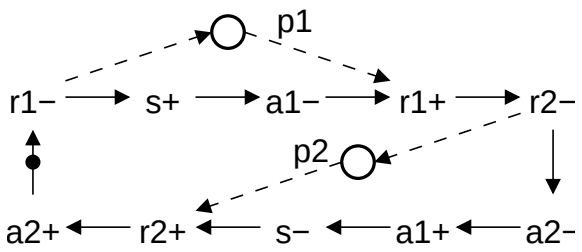
The implementation of PN fragments as DCs is shown in Figure 2. The first three solutions in Figure 2(a) implement linear, join and fork fragments, the last three implement controlled choice, merge and a read-arc. The case of arbitration is special. One of possible implementations for internal arbitrated choice is shown in Figure 2(b). Symbols 'c', 'or' and '&' in drawings of DCs denote logic operations performed on the corresponding inputs. These functions are Muller C-element function for 'c', logic OR for 'or' and logic AND for '&'. Figure 3 shows gate-level implementations for such cells.

In this paper we do not discuss more general cases involving non-persistent transitions and input signals. The reason for this is that signals with non-persistent excitation may create hazards or cause metastability. Design of modules using such signals is still an art often requiring manual analyses and analog simulation [21, 1, 3]. After such a module has been developed, its outputs can be viewed as inputs for the part of the design that uses a direct mapping method.

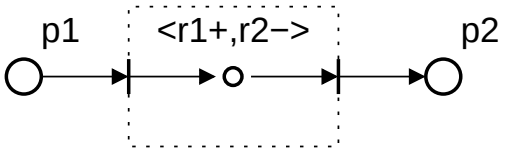
Faster and more compact solutions for a DC implementation can be designed by introducing timing assumptions. Though circuit-level refinement techniques are not the focus of this paper, we show some of DC implementations we developed to give the reader an idea about one of possible sources to our performance gains (cf. case study section). In all DC implementations shown in Figure 4 the reset phase of state holding elements happens concurrently with the move of a token into the next stage DC. This results in a *relative timing* assumption of the reset phase being fast enough in order to preserve the correctness of circuit operation, which should be verified at the implementation stage. These circuits exhibit the following latency in the four-DC ring benchmark



(a) Circuit



(b) STG



(c) PN modelled by a DC

Figure 1: David cell

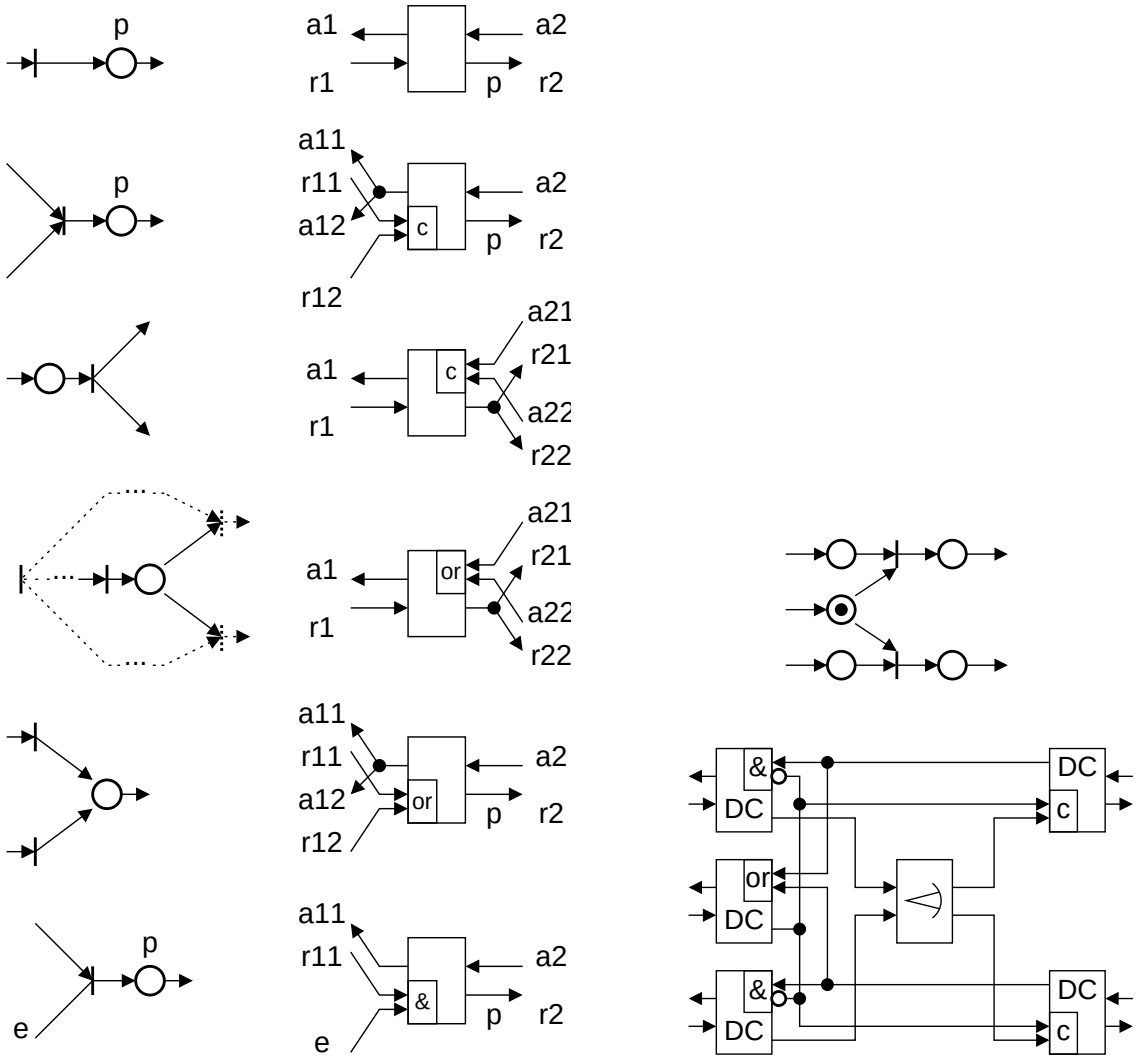
implemented in AMS-0.6μ technology: (a) 0.29ns; (b) 0.17ns; and (c) 86ps. These values, obtained by SPICE simulation, are significantly better than the latency of the original DC shown in Figure 1(a), which equals to 0.99ns. An interesting feature of the last two implementations is that they *internally* contain GasP interface [22], which uses a single wire to transmit a request in one direction and an acknowledgement in the other. Dashed lines in Figure 1(b-c) show where the circuit should be split so that the wire connecting its parts implemented GasP protocol. Combining protocols in a single cell facilitates floorplanning of the design. Short interconnect can be efficiently implemented in GasP, saving area and increasing speed of the block, and long interconnect (connections between blocks) can be implemented as a delay-insensitive [23] handshake on two wires, which guarantees correct operation and facilitates design reuse.

3 Device-environment interface

3.1 Overall approach to interfacing in direct mapping

The only STG component which is directly related to inputs and outputs is the labelling function λ. It binds transitions of the underlying PN to signals, more precisely to signal transitions. A transition in a traditional DC structure is represented as a handshake, which involves two wires ('request' and 'acknowledgement'). The first, straightforward approach to bundle an external signal to a DC structure is to use a converter that transforms an input signal transition (e.g. $x+$) into a handshake event (e.g. $r_{x+}+ \rightarrow a_{x+}+ \rightarrow r_{x+}- \rightarrow a_{x+}-$) or an output handshake event into a signal transition. This is similar to the classical Mealy model of a finite state machine (FSM), in which outputs are bound to transitions.

An alternative is to bind environment interface to the local states (PN or STG places), which resembles the Moore model of an FSM. To use this type of interfacing one has to transform the system specification in such a way that dedicated places were associated with the states (logic



(a) Persistent transitions

(b) Arbitrated choice

Figure 2: DC implementation of PN fragments

values) of interface signals and *only read-arcs* were used for interfacing between the device and environment models. This approach is more adequate than the first one in modelling signals in physical wires connecting a device to its environment.

Environment tracking. Our direct mapping method uses the idea of mutual 'tracking' between the device and the environment. The device model Md, as shown in Figure 5, contains the model of the system Ms', possibly incomplete as it 'hides' internal actions of the environment. It is synchronised to the environment using input signals. The environment model Me also contains the model of the system Ms'' (excluding the internal actions of the device) and uses the outputs of the device to synchronise its own operation. In the language of STGs this idea is realised by constructing two copies of the original specification representing a device and an environment, connected by read-arcs.

A simple example of such an interaction is shown in Figure 6(a). This figure shows the result of the copying. The inputs of the device are outputs for the environment and conversely. The successor place of an output transition (called *interface place*) is interpreted as an output signal

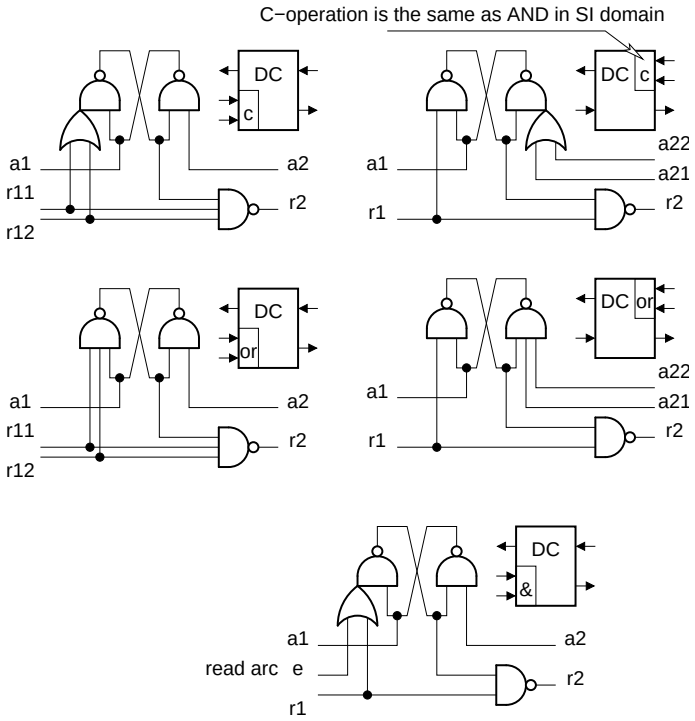
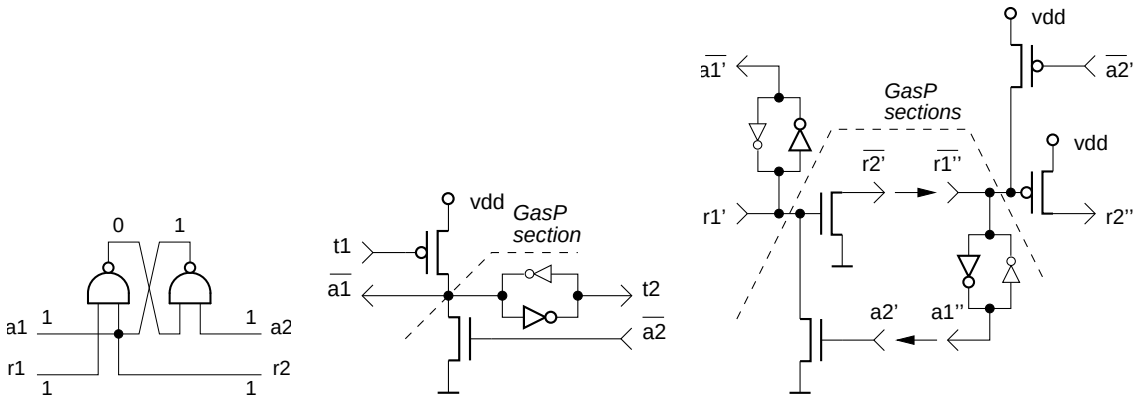


Figure 3: Gate-level implementation of DCs



(a) DC with timing assumptions

(b) Transistor-level implementation

(c) Two DCs implemented as negative gates

Figure 4: DC implementation with timing assumptions

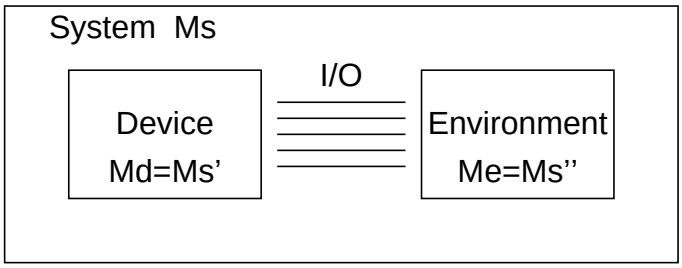
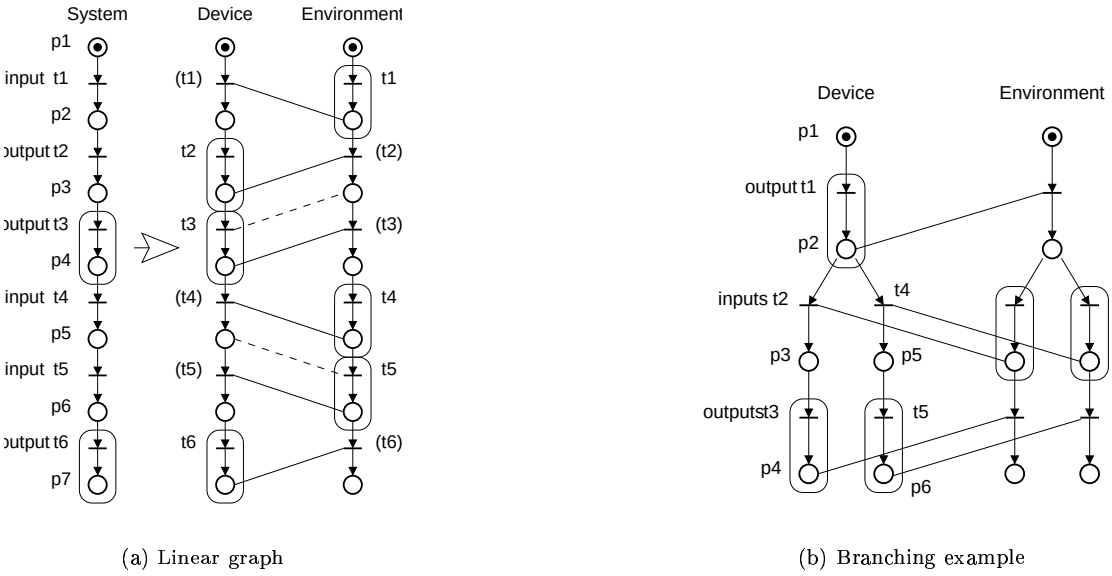


Figure 5: Device-environment interaction

(in Hollaar's style), these are encircled in ovals. It is connected to the corresponding input transition of the opposite part of the system by a read-arc. Input transitions become 'dummies' denoted by parentheses. For example, the input event $t1$ takes place in the environment and it is followed by the tracking dummy transition $(t1)$ in the device. Full tracking involves all pairs of corresponding events synchronised by both solid and dashed read-arcs. This guarantees *observation equivalence* or *weak bisimulation* [15] w.r.t. the input/output transitions. Another example in Figure 6(b) shows the device-environment decomposition of a system with free choice, where choice is made in the environment. It is important that free choice is converted in the device model into deterministic choice controlled by the environment. The obtained model is again equivalent to the original specification by weak bisimulation.



(a) Linear graph

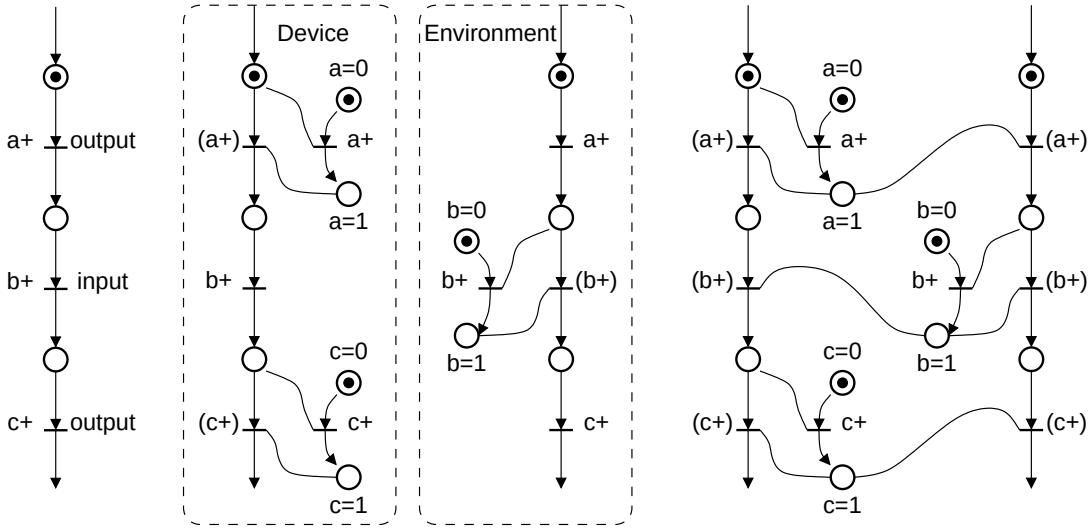
(b) Branching example

Figure 6: Tracking

Dashed arcs in Figure 6(a) represent a potential implementation problem. They control synchronisation of consecutive inputs or outputs, which cannot be physically realised without timing assumptions (these read-arcs cannot be mapped onto a corresponding input).

The previous construction allows us to interconnect the device to the environment by means of read-arcs, but it requires each part of the system to have access to the internal states of its counterpart (via interface places). Indeed, we associated outputs with the postset places of output transitions (Hollaar's style). The physical realisation of the direct mapping approach requires that every interface place corresponded to the logic level of a particular interface signal. For this, we make outputs explicit as illustrated in Figure 7. This transformation is performed by associating with each signal, e.g. a, a pair of complementary places, $a = 0$ and $a = 1$, and connecting them to transitions $a+$ and $a-$. Such elementary cycles are connected to the device and environment nets by using read-arcs as appropriate (cf. circuit PNs [10]). This transformation guarantees STG consistency and, again, preserves weak bisimulation.

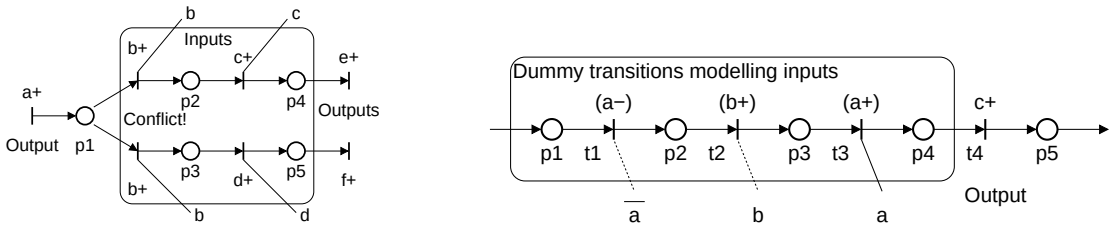
Coding conflicts. It is widely accepted that state coding conflicts do not exist in direct mapping methods. This is only true if all signals of a system are formed inside the device. However, if a device interacts with the environment, then a coding problem may occur. An example in Figure 8(a) shows a device model attempting to track a free choice in the environment. This choice starts one of two sequences $[b+, c+\rangle$ or $[b+, d+\rangle$. Being perfectly legal for the environment,



(a) Spec. (b) Exposition of output signals (c) Coupling

Figure 7: Tracking by explicit outputs

this creates a coding conflict in the device as it cannot distinguish between the choice options at the beginning of these sequences since the binary states of inputs are identical in $p2$ and $p3$. Another example of a coding conflict in the device specification is shown in Figure 8(b). It is also caused by a sequence of unacknowledged inputs $[a-, b+\rangle$, whose tracking is not possible without timing assumptions. If timing assumptions are not allowed, then the dotted arcs should be excluded from the model. In this case, however, a scenario is possible where $p1\,[(a-), (b+)\rangle\, p3$ takes place before $a-$ happens in the environment. Then $t3$ will fire in error as $a = 1$ in both $p1$ and $p3$.



(a) Free choice in the environment (b) Transitions without acknowledgement

Figure 8: Coding conflicts in direct mapping

In order to avoid this kind of problems we apply an additional restriction to the initial specification. The specification *must* be *delay insensitive* to inputs [13], which means that there can be an arbitrary finite delay between the moment of signal transition in the environment and the moment this transition becomes sensed in the device. If one input transition follows another without an output transition in between, then there is always a possibility of inverting the order of how these transitions are sensed. A solution to this problem is to define such inputs as concurrent to each other, thus forming an input *burst* [19]. The task of modifying the initial specification into a burst mode like model and its verification does not belong to the subject of this paper.

From now on, we will assume that the specification does not contain multiple input transitions

positioned in sequence.

3.2 Direct mapping with explicit interfaces

After the initial STG has been expanded to the form in which interface signals between the device and environment are made explicit (represented by pairs of complementary places whose marking corresponds to the logic level of the signal), it is ready for syntax directed mapping to a circuit. For such mapping we only consider the subnet associated with the device (see Figure 7(c)) and those read arcs which connect the input tracking transitions with their appropriate places in the environment subnet. We assume that the environment behaves according to its subnet, which allows us to consider the target circuit as an *open system*, as opposed to *autonomous circuits* in [14]. Structurally, the mapping has three main parts:

1. The main control structure of the device, which tracks the protocol in its internal states and dummy events.
2. The output flip-flops, which directly implement the explicit output places.
3. The input wires, which implement read arcs coming from the environment subnet.

The way of implementing the first part follows the DC-based method shown in Figure 2. Below we consider how the second and third parts are realised. The main criterion for implementing outputs and inputs is concerned with the overall system performance. This is achieved in our technique by, first, minimising the latency of producing an output and reacting to an input, and, second, by maximising concurrency between switching interface signals and performing internal actions in DCs.

Outputs. To illustrate the implementation of outputs we use an example shown in Figure 9. The states of the output y are made *exposed* in places y and $\bar{y}$. The transition $y+$ of the specification is replaced by a 'dummy' $(y+)$, and the actual transition is moved into the model of the signal, where it transfers the token from $\bar{y}$ to y. The enabling function of $y+$ is preserved by inserting read-arcs between the dummy preset $\{p1, p2\}$ and $y+$. An additional read-arc, between y and the dummy, is required to guarantee the persistence of the $y+$ excitation. The pair of places $\langle \bar{y}, y \rangle$ and transitions between them can be implemented as a flip-flip. The advantage of such an implementation is that the output latch is directly controlled by the internal states $\{p1, p2\}$, which minimises the latency of the output. The dummy transition $(y+)$ fires after $y+$, which makes it concurrent to the critical path in the environment.

The DC implementation of the above example is shown in Figure 10(a). Fast versions of DCs (such as those shown in Figure 4(c)) are used to create a transistor-level implementation of the same circuit shown in Figure 10(b). STG places in that schematic are modelled as *keepers* (cross-coupled inverters), the output latch is also implemented as a pair of cross-coupled inverters with set and reset transistor meshes. The latency of $y+$ transition in this example is determined by a single logic gate (transistors $Ts1$ and $Ts2$), which is fast. In AMS-0.6μ technology the latency of $y+$ is 0.14ns without load.

Inputs. Inputs are connected to the main control structure following the idea of Figure 7, using read arcs. The logic implementation of input connection (via read-arcs) should be clear from Figure 3.

To illustrate the fast transistor-level implementation of inputs we use an example in Figure 11, which is mapped into the circuit shown in Figure 12

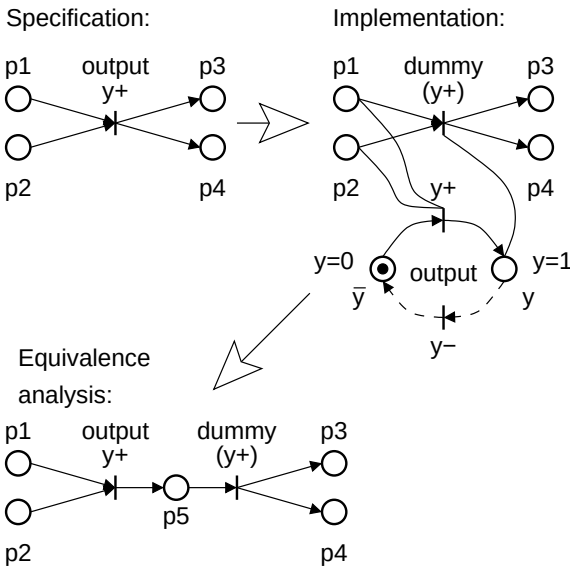
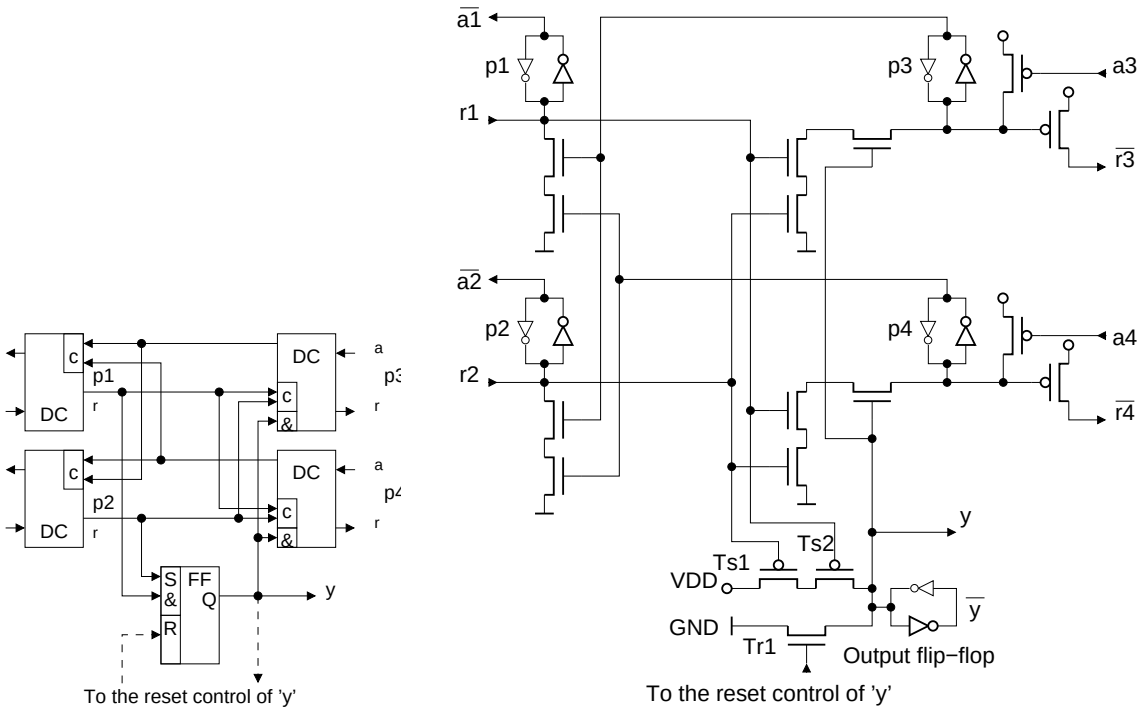


Figure 9: Output exposition



(a) Circuit

(b) Fast transistor-level implementation

Figure 10: DC implementation of the output

The input latency in this circuit is minimal as transistors $Tx1$ and $Tx2$ directly control latches $p3$ and $p4$ modelling the postset of the input transition.

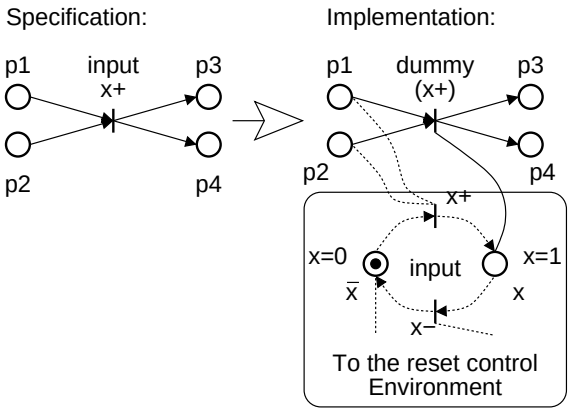


Figure 11: Inputs as local states (STG places)

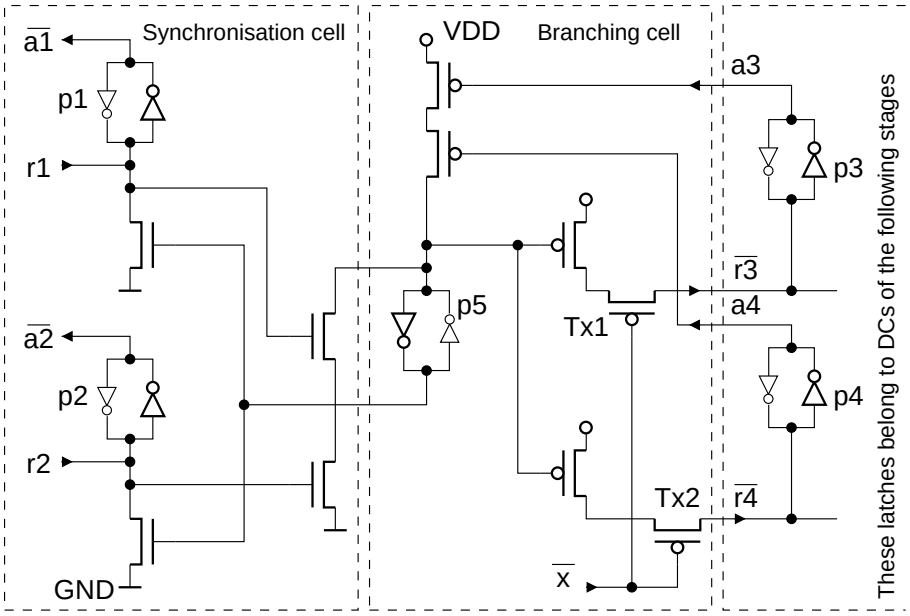


Figure 12: Example of transistor-level implementation of an input

4 Refinement

In the previous section we have shown a canonical way of direct mapping, which converts *all* input and output transitions to their dummy 'images'. Many of these 'dummies', and internal states associated with them, can be removed by the following refinement method. For simplicity, let us consider a linear fragment of an STG specification in Figure 13.

At first, the interface signal exposition is performed, which introduces an elementary cycle for each input/output.

The second step eliminates some of the dummies, which allows input and output signals to participate directly in the control of outputs. In other words, instead of always chaining 'output-dummy-input-dummy-output-dummy ...' we connect, wherever possible, output flip-flops to the previous signal, input or output, directly. The condition for this reduction is that only those signals whose switching directly precedes the given output transition are used in the support of the output (the principle of *locality*). This differentiates our direct mapping approach from logic synthesis techniques, which normally use all inputs to form support for an output. The principle of locality is crucial here because it facilitates the design of faster output flip-flops by minimising

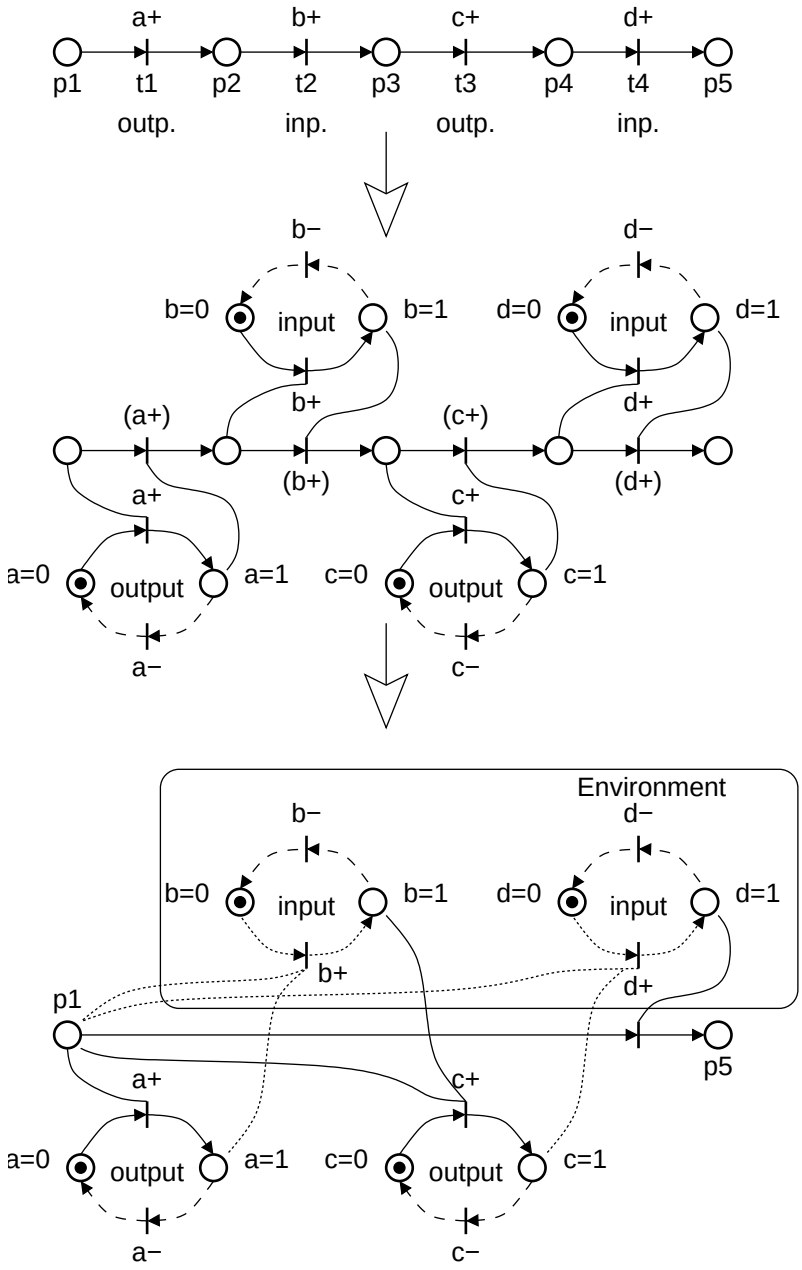


Figure 13: Reduction technique

their (set and reset) fanins. Although the motivation for such a reduction is obvious, it may not always be possible due to potential *coding conflicts*, as explained below.

The dummy elimination, illustrated in Figure 13, results in the $p1$ and $p5$ places and two flip-flops for outputs a and c. The flip-flops are controlled directly by inputs, which minimises latency.

One can also see that $p1$ and $p5$ have a role of 'brackets', separating the given STG fragment from the remainder of the specification, for this reason they are named *subnet identifiers*. The latter play a key role in providing *state separation* between subnets as shown below.

To illustrate the effect of dummy elimination on state separation, consider an example shown in Figure 14. The places $p1$ and $p4$, labelled with '!!!', correspond to the states where $a = 0$. If we eliminate all the dummies and their internal states (DCs), the enabling functions for outputs $b+$ and $c+$ will be identical, namely $p1 \wedge \overline{a}$, which is obviously wrong. To prevent such a coding

conflict we must preserve at least one place on the path between $a+$ and $a-$. For example, place $p3$ was chosen to be such a state separation place.

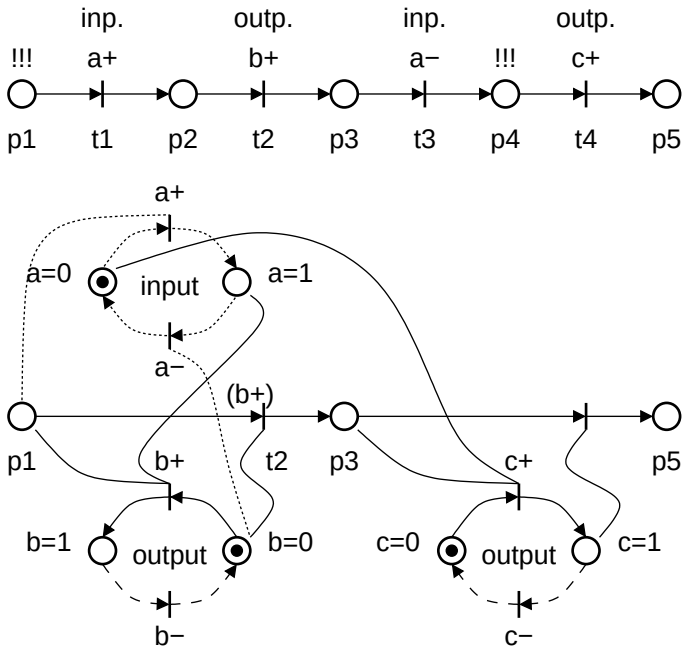
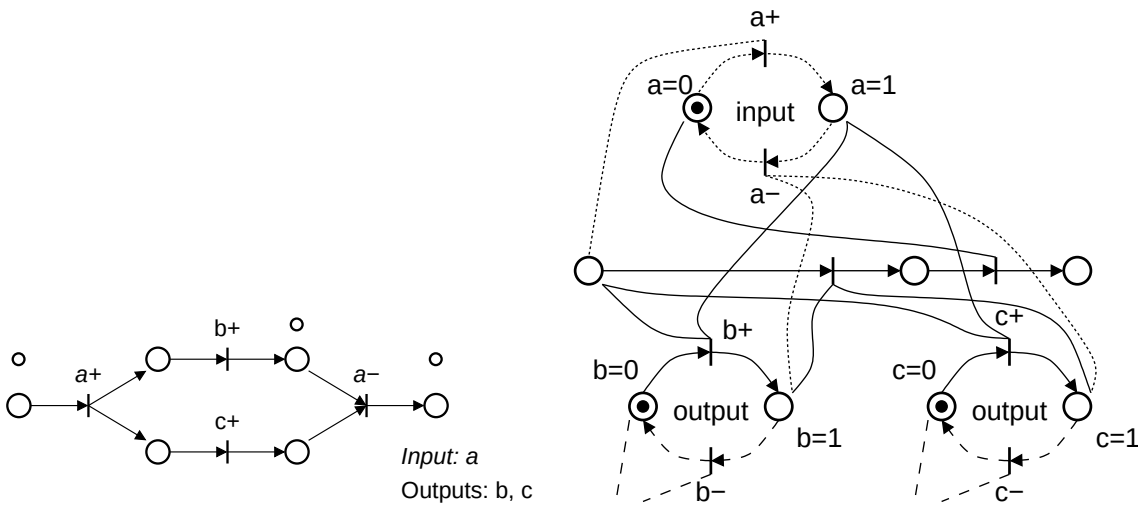


Figure 14: Local coding conflict for an output

The algorithm for determining the state separation places is illustrated in Figure 15. It performs elimination of internal places (one place at a time) and corresponding dummies, every time checking for the local coding conflict. The order in which places are eliminated can be guided by optimisation criteria such as circuit size and performance. In the given example the places tagged with small circles solve the state separation problem. Ultimately, the faster solutions are generally those, where state separation places either follow output transitions or precede input transitions.



(a) STG fragment

(b) State separation

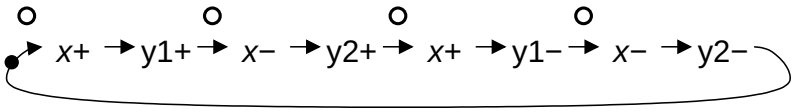
Figure 15: Finding state separation places

It should be noted that the state separation algorithm also includes the analysis of the presence

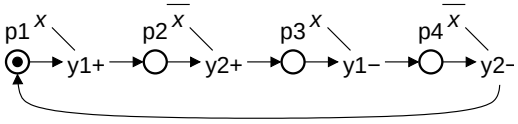
of *takeovers* [14] in STG, which affects the dummy elimination procedure. Similarly, optimisation heuristics are left outside the scope of this paper.

5 Case study

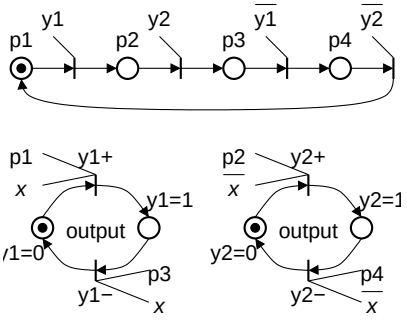
It is widely accepted that direct mapping methods are inefficient in implementing small designs, giving excessively large and slow solutions. We have tested this 'postulate' using a small benchmark, the 'toggle' element, whose STG is shown in Figure 16(a). State separation places in this figure were determined by a sequence of attempts to remove places starting from the initial marking. Those, whose removal led to the local coding conflicts were tagged with small circles. Figure 16(b) shows the result of the separation of the device tracking model from the environment's one and a subsequent elimination of the places which are not needed for state separation. In Figure 16(c) the output signals are exposed in elementary cycles connected to the tracking structure by read-arcs. At this stage the specification is ready for direct mapping.



(a) State separation places



(b) Separating device from environment



(c) Exposition of outputs

Figure 16: Toggle element

The result of the direct mapping of the STG of Figure 16(c) into fast DCs and dynamic flip-flops is shown in Figure 17. Four DCs of alternating (positive-negative-positive-negative) polarity are used to implement the tracking part of the specification. Two flip-flops at the bottom of the circuit diagram perform latching of the outputs. Input signal x, in this implementation, directly controls these flip-flops. This results in the latency of only one gate delay under the assumption that the tracking part is faster than the environment. This assumption is quite reasonable – a single step of tracking involves switching a single DC stage, which by its speed is comparable to an inverter.

We compare this toggle implementation to the manual solution of [8], shown in Figure 18(a), and the circuit automatically generated by Petrify tool [5], shown in Figure 18(b). The results of this comparison are given in Table 1. They show that the DC implementation is significantly faster, though it uses twice as many transistors. The speed increase can be explained by two reasons. The first is the depth of logic, which in our approach is either one or two, depending on the use of the input inverter. The second is the use of very fast gates, with small fanin, in the output flip-flops.

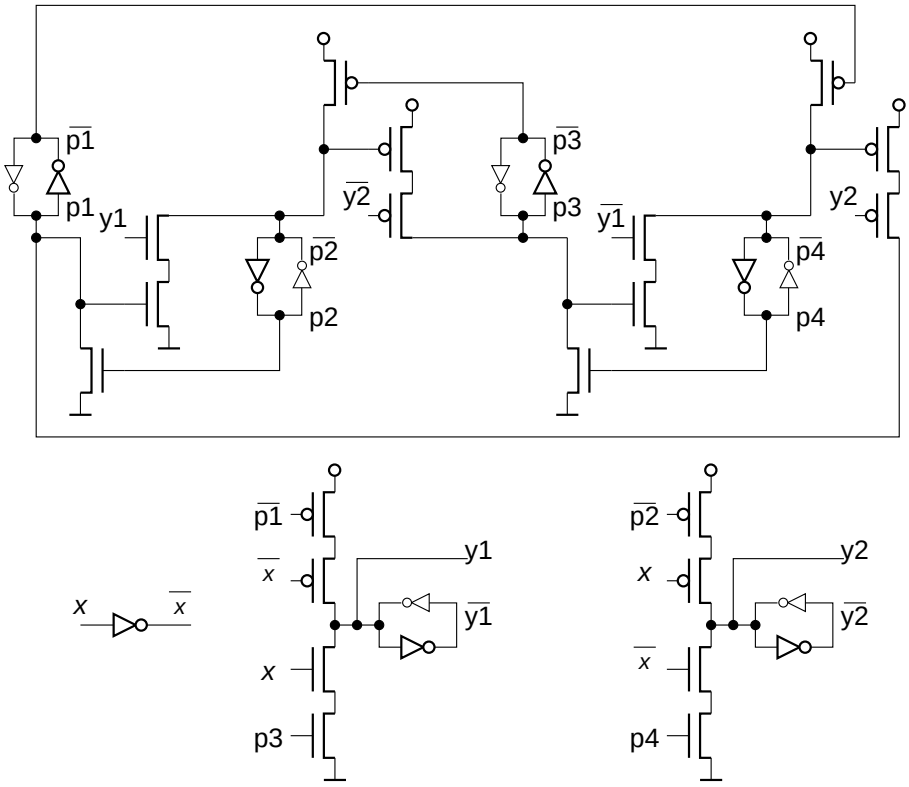
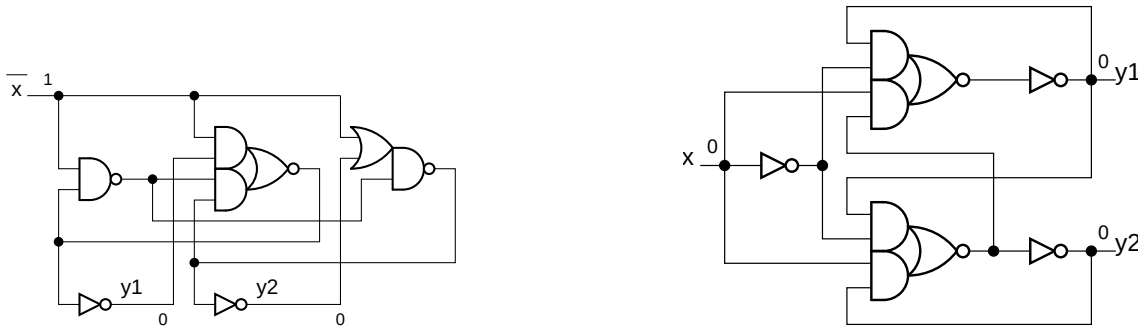


Figure 17: Transistor-level implementation for toggle



(a) Manual design by CSEM

(b) Generated by Petrify tool

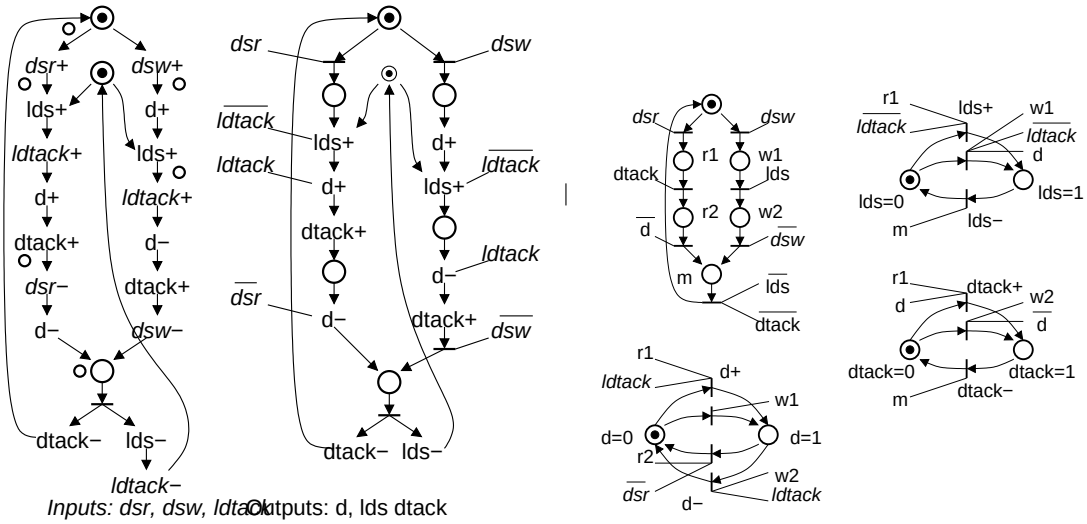
Figure 18: Other toggle implementations

Table 1: Comparison of toggle implementations

Transition	CSEM*	Petrify	fast DC
$x+ \rightarrow y1+$	464ps	289ps	213ps
$x+ \rightarrow y1-$	412ps	567ps	103ps
$x- \rightarrow y2+$	350ps	515ps	148ps
$x- \rightarrow y2-$	413ps	288ps	216ps
Transistors	22	22	46
* Input x has active level 0			

An example in Figure 19 is a specification for a VME bus controller, which is a more complex STG with a non-trivial branching structure. The steps applied to the STG model are similar to

the toggle example. Note that the controlled choice place is not needed for code separation and is eliminated in the final version of the STG. Note also the number of output transitions and read-arcs controlling them in the elementary cycles in Figure 19(b). Every transition in the original STG (Figure 19(a)) has a separate image in an elementary cycle (Figure 19(b)) usually controlled by $n + 1$ read arcs, where n is the number of incoming arcs to this transition in the original STG. Their number can be fewer if the place directly preceding the given transition is not eliminated, but never greater. This allows the designer to estimate the fanin and the latency of outputs in the hardware implementation. For example, $ldtack-$ has a single incoming arc in the original STG and is preceded by a state separation place, hence its image in the elementary cycle has a single controlling read-arc. The transition $lds+$ in the write branch of the original STG has two incoming arcs and no state separation places in the preset, this leads to its image in the elementary cycle being controlled by three read-arcs.



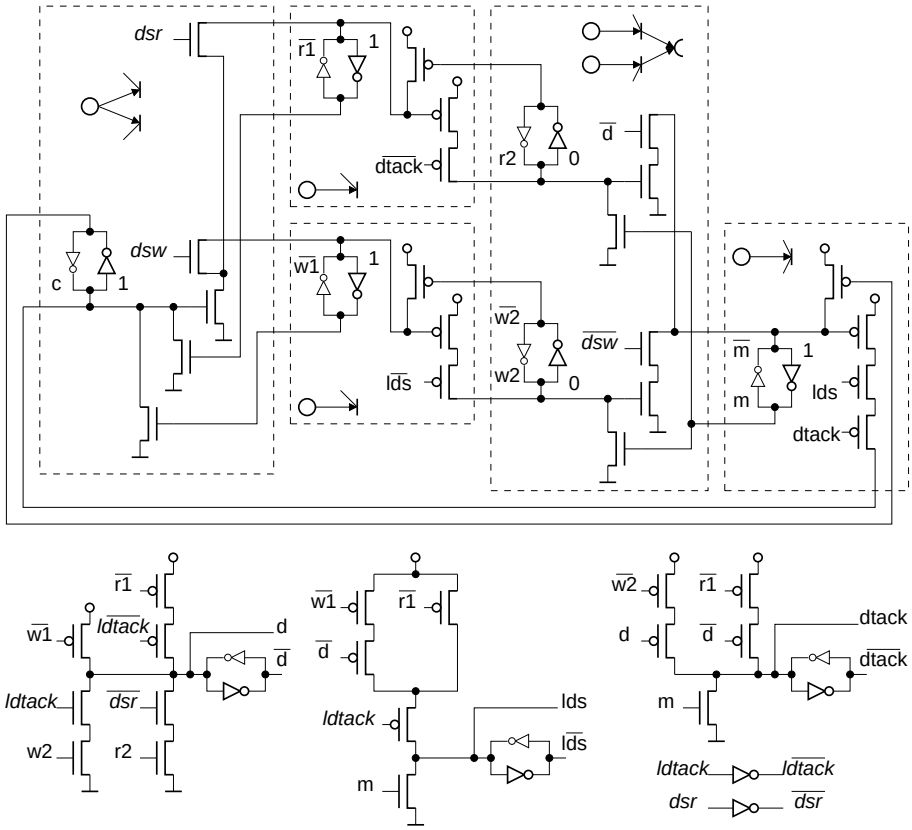
(a) Positioning DCs and separating from environment

(b) Exposition of outputs

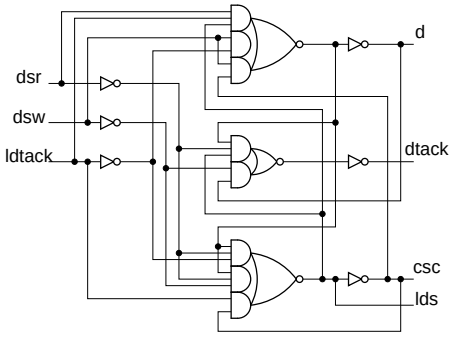
Figure 19: VME bus controller

The result of the direct mapping of the VME bus controller is shown in Figure 20(a). The Petrify solution is shown in Figure 20(b). It can be seen that in the former the output flip-flops are almost always directly controlled by the circuit inputs. In a few cases an inverted input is required, which results in an additional small delay. The reset phases of *lds* and *dtack* and the set phase of *d* (in the write mode) are only controlled by DCs since the direct predecessors of the corresponding transitions in the original specification are selected as state separation places.

These VME bus controller solutions are compared in Table 2, which shows a clear advantage of the direct mapping approach. The longest latency in the Petrify implementation $dsw- \rightarrow dtack-$ is caused by the following path in logic gates: inv+, an332-, an32+, inv-; where 'an332' and 'an32' are slow complex gates. The longest latencies of the direct mapping circuit are due to additional input inverters on some inputs and due to a 'join' construct in the original STG, which leads to higher flip-flop fanin. The numbers of transistors in both circuits are almost the same.



(a) Direct mapping



(b) Generated by Petrify tool

Figure 20: VME bus controller implementations

6 Summary

In this paper a method for direct mapping an STG specification into a control circuit is presented. The method exploits the ideas of output exposition and environment tracking.

Output exposition is a behaviour-preserving (up to weak bisimulation) transformation of the STG that creates a pair of places for the values 0 and 1 of each output signal. Output transitions are only allowed between the places within the corresponding pair. In our method these transitions communicate with the remainder of the STG by means of read-arcs, which facilitates implementation of outputs as flip-flops.

Table 2: Comparison of VME bus controller implementations

Transition	Petrify	Fast DC
$ldtack+ \rightarrow d+$	0.89ns	0.29ns
$ldtack+ \rightarrow d-$	0.94ns	0.16ns
$d+ \rightarrow dtack+$	0.38ns	0.27ns
$dsw- \rightarrow dtack-$	1.19ns	0.26ns
$ldtack- \rightarrow lds+$(read mode)	0.61ns	0.21ns
$ldtack- \rightarrow lds+$(write mode)	0.61ns	0.29ns
$dsw- \rightarrow lds-$	0.64ns	0.26ns
Number of transistors	52	56

The idea of tracking the environment is important as it prevents coding conflicts and simplifies the control of output flip-flops. Tracking by our method is possible if the specification is delay-insensitive w.r.t. inputs. To model the tracking, two copies of the original STG are created, one for the device and one for the environment. These parts are synchronised by read-arcs inserted between input transitions of one part and the explicit output places of the other part. Only the part that corresponds to the device is subsequently implemented as a circuit. The library of David cells is described which can be used to implement the tracking.

The refinement of the method allows the input signals to be used directly in the control of the output flip-flops, thus reducing the latency of the circuit. An output flip-flop is controlled by the internal signal corresponding to the tracking place that directly precedes it and by those input signals whose transitions directly precede the given output transition. This restriction reduces the fanin of the flip-flops and increases their speed. In the process of refinement the tracking components are removed one at a time. Only those are kept in whose removal causes a local coding conflict, i.e. the conflict involving only the set of signals defined by the above restriction.

The design examples exhibit low output latency, which is due to the low depth of logic (one or two gates). This is the characteristic feature of our method. The simulated values of output latency in the circuits created by the direct mapping are significantly smaller than those generated by logic synthesis methods. For small-size benchmarks, the size of direct mapping circuits can be twice as bigger than logic synthesis solutions. This difference becomes less for larger benchmarks.

References

[1] C. H. (Kees) van Berkel and Charles E. Molnar. Beware the three-way arbiter. *IEEE Journal of Solid-State Circuits*, 34(6):840–848, June 1999.

[2] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[3] A. Bystrov, D. J. Kinniment, and A. Yakovlev. Priority arbiters. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 128–137. IEEE Computer Society Press, April 2000.

[4] T.-A. Chu, C. K. C. Leung, and T. S. Wanuga. A design methodology for concurrent VLSI systems. In *Proc. International Conf. Computer Design (ICCD)*, pages 407–410. IEEE Computer Society Press, 1985.

[5] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Petrify: a tool for manipulating concurrent specifications and synthesis of

asynchronous controllers. In *XI Conference on Design of Integrated Circuits and Systems*, Barcelona, November 1996.

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

[7] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[8] E. Vittoz et al. Silicon-gate frequency divider for the electronic wrist watch. *IEEE Journal of Solid-State Circuits*, SC-7:100–104, 1972.

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

[10] J. Grabowski. On the analysis of switching circuits by means of petri nets. *Elektronische Informations-verarbeitung und Kybernetik*, 14:611–617, 1978.

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

[12] D. A. Huffman. The synthesis of sequential switching circuits. In E. F. Moore, editor, *Sequential Machines: Selected Papers*, pages 3–62. 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.

[13] Mark B. Josephs and Jan Tijmen Udding. An algebra for delay-insensitive circuits. In E. M. Clarke and R. P. Kurshan, editors, *Proc. International Workshop on Computer Aided Verification*, pages 147–176. American Mathematical Society, 1991.

[14] Michael Kishinevsky, Alex Kondratyev, Alexander Taubin, and Victor Varshavsky. *Concurrent Hardware: The Theory and Practice of Self-Timed Design.* Series in Parallel Computing. John Wiley & Sons, 1994.

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

[16] U. Montanari and F. Rossi. Acta informatica, 1995.

[17] 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.

[18] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng. Automatic synthesis of gate-level timed circuits with choice. In *Advanced Research in VLSI*, pages 42–58. IEEE Computer Society Press, 1995.

[19] Steven M. Nowick. *Automatic Synthesis of Burst-Mode Asynchronous Controllers.* PhD thesis, Stanford University, Department of Computer Science, 1993.

[20] L. Y. Rosenblum and A. V. 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.

[21] Charles L. Seitz. Ideas about arbiters. *Lambda*, 1(1, First Quarter):10–14, 1980.

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

[23] Jan Tijmen Udding. *Classification and Composition of Delay-Insensitive Circuits.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1984.

[24] S. H. Unger. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[25] V. I. Varshavsky and V. B. Marakhovsky. Hardware support of discrete event coordination. In *Workshop on Discrete Event Systems WODES96*, pages 332–339, Edinburgh, 1996.