# Petrify: a tutorial for the designer of asynchronous circuits

Jordi Cortadella
Department of Software
Universitat Politècnica de Catalunya
jordic@lsi.upc.es

# Contents

# Preface

Petrify is the fruit of several years of common research of a group of people with multidisciplinar expertise. The first line of code was written on December 6, 1994. The main purpose of the first prototype was to demonstrate that some algorithms we devised to synthesize Petri nets were computationally affordable and effective in practice.

After a couple of months, we built our first petrify that was about 2,000 lines of code. The functionality of the program was simple: it could read a Petri net, perform reachability analysis and produce another "simpler" Petri net with bisimilar behavior.

After the first results, we all got excited about the wide spectrum of applicability that the tool could provide. It was then when we realized that petrify could be a useful tool in the synthesis flow of asynchronous circuits. And we started adding functionality to the tool.

The next significant contribution was solving the state encoding problem of asynchronous specifications. By that time, spring 1996, the tool was about 20,000 lines of code. After solving that problem, we felt that we had almost all we needed to complete the path from specification to circuit.

Finally, we solved the problem of logic decomposition and technology mapping. At that point, it was crucial the participation Enric Pastor, who provided most of the code for technology mapping based on boolean matching, and few more thousands of lines of code.

During the last year, we decided to escape from the speed-independent world and incorporate timing assumptions in our design flow. In that respect, we must thank the help and feedback received by some colleagues at Intel Corporation that hosted my visit during the summer of 1998. I would like to specially Shai Rotem, Ken Stevens, Marly Roncken and Ran Ginosar.

Today, petrify is about 50,000 lines of C code. Unfortunately, not as well-structured as one would like, since petrify has often been written with the pressure of some deadline to obtain results and write a paper. And without the pressure of being a commercial product that should friendly interact with the user.

But "the important is inside" and petrify's "inside" contains many methods, algorithms, data structures and heuristics that were jointly devised by a greographically distributed team of friends. Most of the intellectual richness of petrify has been published in several papers. That is why this document will give a different view of the tool: a pratical view.

But I would not like to finish this preface without mentioning the team of colleagues with whom I spent so many days of exciting discussions and so many nights of ...

*Mike Kishinevsky*, who was always able to find the best way of explaining in simple words the most confusing concepts in our minds. *Alex Kondratyev*, who was always able to prove that theorem that nobody could prove and to find a counterexample to our simplest

conjectures. *Luciano Lavagno*, who always knew where we were, even in the most confusing and depressing moments of our discussions. *Enric Pastor*, who saved our souls when implementing technology mapping seemed like an unreachable summit. *Alex Taubin*, who always enriched the discussions with his personal phylosophical view of life. And *Alex Yakovlev*, who was always able to write three pages about Petri nets in our papers when all the others could write no more than two sentences.

# Chapter 1

# Introduction

When petrify was first created, it was supposed to become a framework for the synthesis of Petri nets. We soon realized that this framework could significantly improve the synthesis flow for the design of asynchronous systems specially in what refers to the interaction between the designer and CAD tools. Finally, petrify has become a tool mostly devoted to asynchronous design with additional support to the synthesis of different classes of Petri nets.

Behind most petrify's features there is a common goal: to enable a fluent interaction with the designer. Even though several CAD tools for the synthesis of asynchronous circuits have recently emerged as the fruit of several years of research in this area, the designer often feels the need to manually interfere in their design flow to produce the desired solution to her or his problem.

While designing petrify we always had, and still have, the following idea in mind: *"telling the designer how a solution has been obtained and how it can be modified is sometimes more important that reporting the solution itself"*. Nowadays, the design of an asynchronous circuit is still an iterative task in which CAD tools cover one of the basic steps in the design flow. But in this iterative process, several solutions are usually generated until the designer feels that the quality of the circuit cannot be significantly improved.

The solution provided by a CAD tool at each step is often not the one expected by the designer. There are many reasons that can explain this phenomenon:

- The methods implemented in the tool mostly face problems of unmanageable complexity, i.e. belonging to the NP-complete or NP-hard categories. Therefore, heuristics that only explore a small fraction of the solution space are tipically devised and suboptimal solutions are generated.

- The designer does not always have a clear idea of the critical paths of the circuit until a solution is reported and simulated.

- Asynchronous systems are usually composed of smaller subsystems. Each subsystem acts as environment for the other subsystems. Thus, finding a global solution for a system may require several iterations in which different subsystems are tuned according to their mutual interaction.

For all the previously mentioned reasons and whenever possible, petrify tries to avoid reporting an obscure, although correct, solution in which the designer completely ignores how to progressively improve it.

### What you should know to read this tutorial

We assume the reader to have some background on the design of asynchronous circuits. Petrify uses Signal Transition Graphs (*STG*s) as a formalism to specify the behavior of the circuits. *STG*s are Petri nets in which transitions are interpreted as signal transitions. For this reason we also assume the reader to have some basic knowledge on Petri nets and on logic synthesis.

The actual syntax used to described *STG*s and gate libraries is the one used in SIS. Although is not crucial to follow this tutorial, we recommend the reader to look at the documentation provided with SIS to become familiar with the astg and genlib formats.

### What you will find in this tutorial

This tutorial is an attempt to show the basic features of petrify with simple examples. The tutorial does not cover the whole functionality of petrify. For that, you can read the manual page provided with petrify that explains all the options of the tool and the extensions of the astg format.

If you become an expert designer and are not satisfied with the current information provided for the tool, you can always contact the author of this document for further details and help.

The first chapter of this tutorial is devoted to the synthesis of Petri nets. The reader might wonder why a tutorial on asynchronous circuit design wastes a chapter in topics not relevant to this field. Petri net synthesis is the key engine for backannotation and interaction with the designer. You will soon realize the importance of this topic and how a basic knowledge on petrify's capabilities for Petri net synthesis can be crucial to understand the transformations performed on the circuit specification.

### What you will not find in this tutorial

This tutorial is aiming at explaining *what* petrify does and not *how* it does it. It is mainly oriented to designers rather than CAD developers. For an exhaustive list of publications explaining *how* petrify works solving different synthesis problems, we refer the reader to `http://www.lsi.upc.es/~jordic/petrify/refs`.

### Where the tool is

Petrify has been compiled for different platforms (including MS-Windows 95). The tool and related information can be found in `http://www.lsi.upc.es/~jordic/petrify`.

# Chapter 2

# Petri net synthesis

## 2.1 Specifying and displaying Petri nets

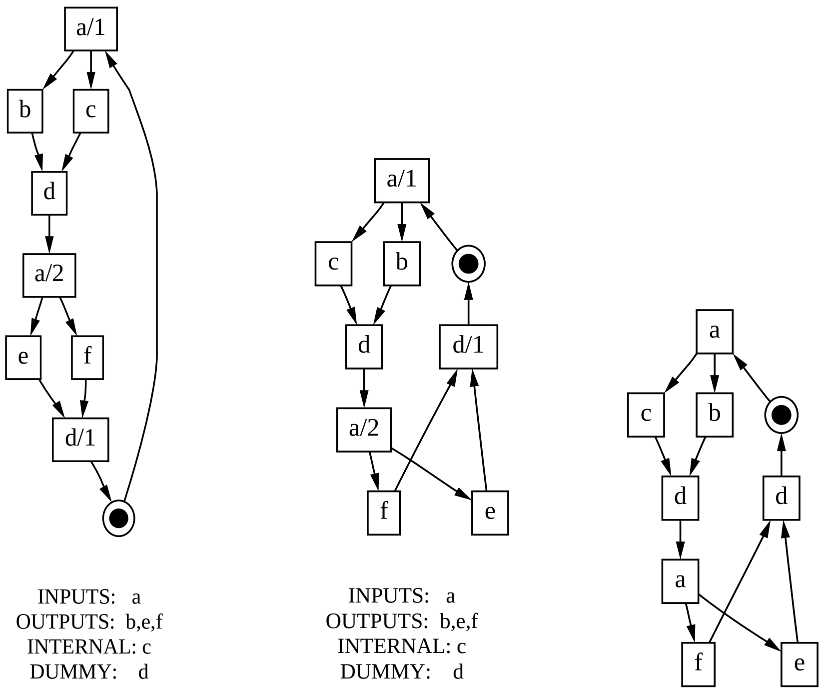
Figure 2.1(a) depicts the specification of a Petri net. Initially, sets of signals are declared. Input and output signals are supossed to be the signals observable by the environment. Internal signals are used to specify non-observable behavior. Symbols declared as “dummy”

```
.model pn_synthesis

# Declaration of signals
.inputs a
.outputs b e f
.internal c
.dummy d

# Petri net
.graph
a/1 b c
b d
c d
d a/2
a/2 e f
e d/1
f d/1
d/1 a/1

# Initial marking
.marking { <d/1,a/1> }
.end
```




(a) (b) (c) (d)

Figure 2.1: (a) specification of a Petri net in astg format (pn_syn.g). (b,c,d) pictures generated by draw_astg.

are only used for synchronization and do not denote any activity of the system.

The Petri net is specified by listing the arcs between places and transitions. When multiple transitions with the same label exist in the specification, indices are used to distinguish tjem (e.g. a/1 and a/2). A label with no index is equivalent to one with index 0.

Places with one input and one output arcs do not need to be explictly declared. An arc from transition to transition is assumed to hold an implicit place. In our example, no explicit places have been declared.

The picture depicted in Figure 2.1(b) has been obtained by the following command:

```
draw_astg -nofold -bw pn_syn.g -o pn_syn.g.ps
```

Draw_astg is a tool that can generate different types of graphical formats to represent *STG*s and state graphs. Tipically it is used to generate postscript files. The option `-nofold` usually generates a picture with longer height, but it sometimes helps to improve the planarity of the layout. The picture obtained without the `-nofold` option is shown in Figure 2.1(c). The option `-bw` generates black-and-white pictures. It is recommended when producing graphics for black-and-white printers.

In color pictures, different colors are used for input, output and internal signals. We suggest the reader to look at the previous layout with the following command:

```
draw_astg pn_syn.g | ghostview -
```

We also suggest to try the command:

```
draw_astg -nonames -noinfo -bw pn_syn.g -o pn_syn.g.nonam.ps
```

that will produce the picture in Figure 2.1(d). The option `-noinfo` suppresses the caption on signal types depicted at the bottom of each picture. The option `-nonames` suppresses places names (none is explicitly declared in this example) and the indices that distiguish different instances of transitions.

## 2.2 Generating the state graph

The state graph corresponding to the previous example can be obtained by the following command:

```
write_sg pn_syn.g -o pn_syn.sg
```

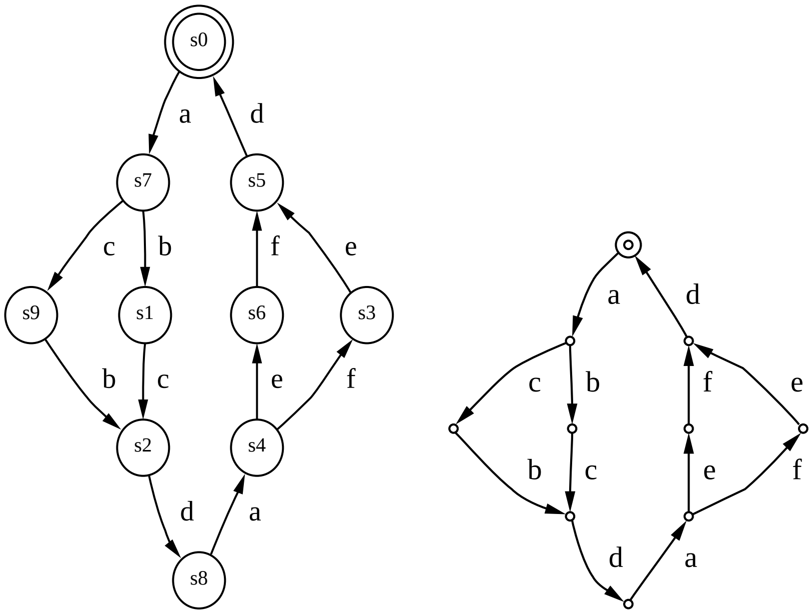
Write_sg is a tool that can be easily combined with draw_astg to depict state graphs as follows:

```
write_sg pn_syn.g | draw_astg -sg -noinfo -bw -o pn_syn.sg.ps
```

or also

```
write_sg pn_syn.g | draw_astg -sg -noinfo -nonames -bw -o pn_syn.sg.ps
```

```
# State graph generated ...
# from <pn_syn.g> on    ...
.model pn_synthesis
.inputs  a
.outputs  b e f
.internal  c
.dummy  d
.state graph # 10 states
s9 b s2
s6 f s5
s3 e s5
s5 d s0
s4 e s6
s4 f s3
s7 b s1
s7 c s9
s2 d s8
s1 c s2
s0 a s7
s8 a s4
.marking {s0}
.end
```




(a) (b) (c)

Figure 2.2: State graph generated by write_sg and depicted by draw_astg.

The result of the previous commands can be seen in Figure 2.2.

The text file shown in Figure 2.2 illustrates a new specification format accepted by petrify: the state graph format. It basically consists in a list of labeled arcs. No transition indices are required to distinguish arcs with the same labels, since they are completely determined by the source and destination states of the arc. This format also accepts the definition of multiple arcs (paths) in one text line. The following lines could be used to define some of the arcs of the example:

```
s0 a s7 b s1 c s2
s2 d s8 a s4 f s3
...
```

## 2.3 Synthesis of Petri nets

One of the most important features of petrify is the capability of deriving a Petri net from a state graph. The resulting specification is place-irredundant. When generating the Petri net, petrify aims at minimizing the transition count by mapping arcs of the state graph with the same label into the same Petri net transition.

Let us try the command

```
.model pn_synthesis
.inputs  a
.outputs  b e f
.internal  c
.dummy  d
.graph
a p4 p3 p1
b p2 f e
c p0
d a
e c b p0
f p2
p0 d
p1 b f
p2 d
p3 c e
p4 b e
.marking { <d,a> <e,c> <e,b> }
.end
```

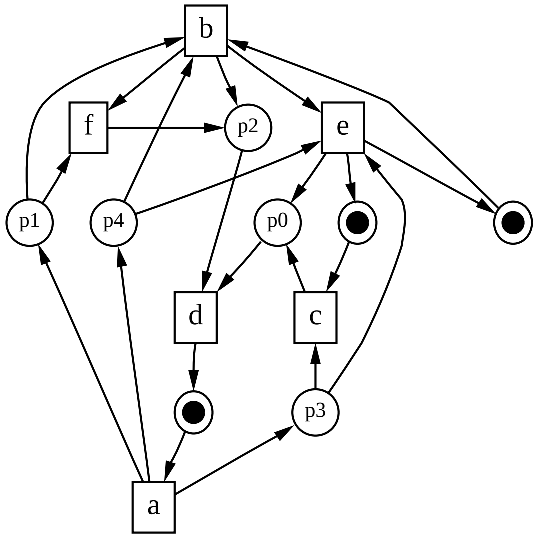



Figure 2.3: Petri net synthesized by petrify (pn_syn2.g).

```
petrify pn_syn.g -o pn_syn2.g
```

The result is depicted in Figure 2.3. Several things can be observed in the picture. First, the structure of the Petri net is apparently more complex than the one of the initial specification and the behavior of the system cannot be easily deduced. Second, the behavior of this new Petri net is "equivalent" (bisimilar) to the original one. Third, unlike the original specification, now there is only one transition for each signal. Finally, the total number of places, including those not explicitly shown in the original specification, is the same in both specifications (10 places).

Usually, the designer prefers to analyze less compact representations but with simpler structures that can help to guess the behavior of the system in a more intuitive way. It is difficult to guide petrify to generate such structures since there is no clear definition of what a "simpler" structure is. Definitely, place an transition count is not a good measure of simplicity in this case, as we have seen in the structure of the Petri net of Figure 2.3.

However, there is an strategy that works in many cases. We can ask petrify to split those transitions whose *excitation region* corresponds to multiple connected sets of states in the state graph[1]. Looking back at the state graph of Figure 2.2(b), we can see that the excitation region of event $b$ is the connected set of states $\{s_7, s_9\}$, whereas the excitation region of event $d$ is the disconnected set of states $\{s_2, s_5\}$. Disconnection often produces different causality relations (predecessor/successor) with the other events. We can ask petrify to split disconneted excitation regions by using the option `-er`. For example:

```
petrify -er pn_syn2.g -o pn_syn3.g
```

[1]The excitation region of an event is the set of states in which the event can fire.

In this particular case, the obtained result would be isomorphic to the original Petri net shown in Figure 2.1.

## 2.4 Transformations during the synthesis of asynchronous circuits

The designer of an asynchronous circuit usually wants that the reported Petri net resembles the original one as much as possible. For this reason, when petrify synthesizes an asynchronous circuit, it tries not to destroy the initial structure of the Petri net. Unfortunately, this cannot be always guaranteed and some of the initial transitions must be split to create a state graph that can be synthesized into a Petri net.

In the rare case that the designer would like to synthesize a circuit and obtain a compact representation of the transformed behavior (e.g. by using only one transition per signal, if possible), two steps would be required: one for logic synthesis and the other for the synthesis of the final specification. For example:

```
petrify [options for logic synthesis] spec.g -o spec_final.g
petrify spec_final.g -o spec_compact.g
```

Petrify also allows to read from the standard input and write into the standard output. Thus, the previous steps can be performed in only one command line:

```
petrify [options for logic synthesis] spec.g | petrify -o spec_compact.g
```

## 2.5 Hiding signals

Specifications may contain signals that are not relevant to the external behavior of the system. In these cases, the designer might be interested in observing the behavior of a subset of relevant signals. Petrify provides a mechanism to hide signals and derive a new specification that is observationally equivalent to the initial one.

This feature is highly interesting to undo some of the transformations that petrify performs when synthesizing asynchronous circuits, e.g. to remove some state signals.

Let us see how we can generate the behavior of our original specification after hiding signals `a`, `b` and `d`. This can be achieved with the following command:

```
petrify -hide .inputs,b,d pn_syn.g -o pn_syn.hide.g
```

The `-hide` option is followed by a list of signals separated by commas. Several `-hide` options can be used in the same command line. The keywords `.inputs`, `.outputs`, ..., can also be used to indicate that the corresponding group of signals must be hidden. The following command would produce the same effect as the previous one:

```
petrify -hide .inputs,b -hide .dummy pn_syn.g -o pn_syn.hide.g
```

```
.model pn_synthesis
.outputs  e f
.internal  c
.graph
e c
f c
c f e
.marking { <f,c> <e,c> }
.end
```

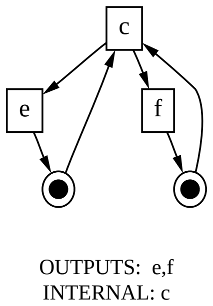



Figure 2.4: Petri net after hiding signals `a`, `b` and `d` (`pn_syn.hide.g`).

The Petri net obtained after hiding the specified signals is depicted in Figure 2.5. Its behavior is equivalent to the original specification if we consider `a`, `b` and `d` to be silent events.

Petrify always hides all `dummy` events when an asynchronous circuit is to be synthesized.

# Chapter 3

# Synthesis of speed-independent circuits

We are now prepared to synthesize our first circuit. Let us take the popular `xyz` example shown in Figure 3.

The picture of the state graph shown in the right has been obtained with the following command:

```
write_sg -bin xyz.g | draw_astg -bin -bw -o xyz.sg.ps
```

The option `-bin` for write_sg and draw_astg indicates that information about binary encoding of the states must be generated by the former and shown by the latter. If you are curious about how this information is exchange, you just have to generate an output file with `write_sg -bin` and have a look at it. The order of the signals in the binary vectors correponds to the order of the signals in the caption of the picture, i.e. first the inputs ($x$) and then the outputs ($y$ and $z$).

```
.inputs x
.outputs y z
.graph
x+ y+ z+
z+ x-
y+ z-
x- z-
z- y-
y- x+
.marking {<y-,x+>}
.end
```

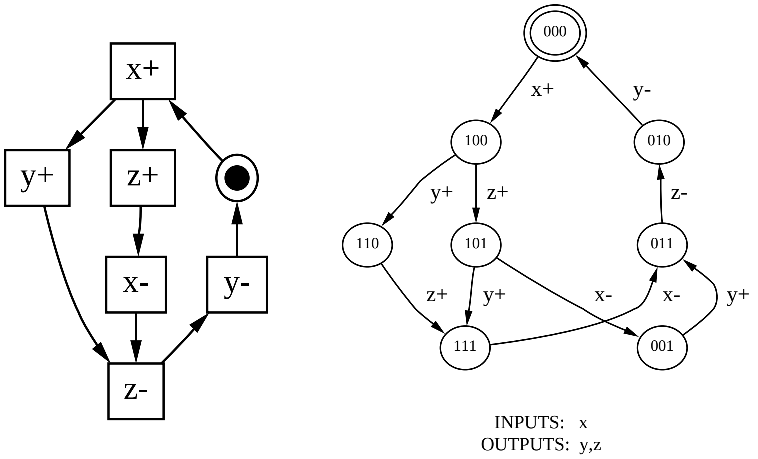



Figure 3.1: The `xyz` example.

## 3.1 Synthesis of complex gates

Let us frist try to generate a circuit in which each non-input signal is implemented as a complex gate. We can obtain that as follows:

```
petrify xyz.g -cg -eqn xyz.eqn -no
```

The option `-cg` indicates that a complex-gate implementation is to be derived. The option `-eqn xyz.eqn` indicates that an output file named `xyz.eqn` in EQN format[1] must be generated. The option `-no` indicates petrify not to generate an output specification. We use this option to avoid reporting a new specification that would be equal to the original one, since no transformations are required in this example to derive complex gates.

The contents of `xyz.eqn` is the following:

```
# EQN file for model xyz
# Generated by petrify 4.0 (compiled 10-Oct-98 at 5:46 AM)
# Outputs between brackets "[out]" indicate a feedback to input "out"
# Estimated area = 10.00

INORDER = x y z;
OUTORDER = [y] [z];
[y] = z + x;
[z] = y' z + x;
```

The number of literals of the equations can be calculated as the reported estimated area divided by 2. Currently, the estimated area reports the transistor count of the circuit assuming that the inverters at the fanin of each complex gate have no cost.

Petrify can also generate its output in BLIF format[2]. The required option for that is `-blif filename`. Both `-eqn` and `-blif` options can be used in the same command line.

## 3.2 The file `petrify.log`

The EQN and BLIF files contain only information readable by SIS. Indeed, the solution to implement a non-input signal is not unique, and the designer might want to explore different solutions and select one according to some criteria. It is for this reason that petrify generates some additional information in a file called `petrify.log`.

Let us look at the contents of the file generated for the previous example:

[1] This is a format for boolean equations used in SIS.
[2] Another format used in SIS.

```
--------------------------
| Input -> Input Delays: |
--------------------------
Average delay =  1.50 events
Worst-case delay =  2.00 events
Input events with worst-case delay: x+
  > Input events preceding x+: x-(2)
  > Input events preceding x-: x+(1)


======================
# Gates for signal y #
======================

y' = x' z'
[y] = y'        (output inverter)
  > triggers(SET):     z- -> y-
  > triggers(RESET):   x+ -> y+
  > 6 transistors (3 n, 3 p)
  > Estimated delay: rising = 19.58, falling = 24.96
  > Speed independent (no timing assumptions)

y = z + x
  > triggers(SET):     x+ -> y+
  > triggers(RESET):   z- -> y-
  > 8 transistors (4 n, 4 p)
  > Estimated delay: rising = 19.96, falling = 26.38
  > Speed independent (no timing assumptions)

======================
# Gates for signal z #
======================

z = y' z + x
  > triggers(SET):     x+ -> z+
  > triggers(RESET):   (y+,x-) -> z-
  > 10 transistors (5 n, 5 p)
  > Estimated delay: rising = 19.96, falling = 26.62
  > Speed independent (no timing assumptions)

z' = x' (z' + y)
[z] = z'        (output inverter)
  > triggers(SET):     (y+,x-) -> z-
  > triggers(RESET):   x+ -> z+
  > 10 transistors (5 n, 5 p)
  > Estimated delay: rising = 19.58, falling = 29.71
  > Speed independent (no timing assumptions)
```

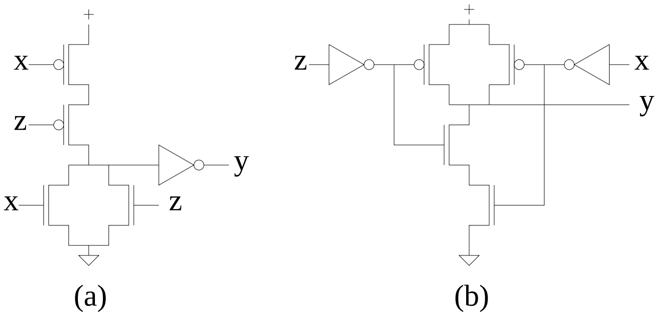



Figure 3.2: Two different implementations of signal $y$.

We will focus now in the information provided for the generation of the complex gate for signal $y$. Two different implementations are generated. The first one is:

```
y' = x' z'
[y] = y'        (output inverter)
```

These equations indicate that $y$ can be implemented with a static gate with a pull-up network implementing $\bar{x}\bar{z}$. The pull-down network is the corresponding dual one. Finally, one inverter must be added at the output (`[y] = y'`). We also have information about the events that trigger the pull-up and pull-down networks. For the designers interested in transistor-level details, this information can also be relevant. Trigger signals are the latest to arrive at the gate. In the case of chains of series transistors, those corresponding to triggers signals should be put close to the output so that the performance of the gate is improved. According to this criterion, the gate depicted in Figure 3.2(a) would be designed.

For the other implementation,

```
y = z + x
```

the gate shown in Figure 3.2(b) would be derived, i.e. no output inverter at the output but with input inverters at the inputs to implement the pull-up network $z + x$. Again, the transistor with gate $\bar{z}$ in the pull-down network should be put closer to the output since the event $z-$ is triggering $y-$.

We can tell petrify to generate a different file name for the `log` file. This is achieved with the option `-log filename`. We can also tell petrify not to generate the `log` file by using the option `-nolog`.

## 3.3 Synthesis of generalized C elements

At this point, we assume the reader to be familiar with the concept of generalized C element (gC). Petrify can also generate equations for gCs as follows:

```
petrify xyz.g -gc -eqn xyz.eqn -log xyz.log -no
```

In `xyz.eqn` we will obtain the following equations:

```
[y] = x + z;
[1] = y x';
[z] = z [1]' + x;          # mappable onto gC
```

We see here that signal $z$ can be implemented as a gC. Still, the designer might not find it trivial to figure out how the gC must be laid out. However, this information is much more readable in the `log` file. For signal $z$, petrify reports the following implementations:

```
======================
# Gates for signal z #
======================

SET(z)    = x
RESET(z) = y x'
  > triggers(SET):   x+ -> z+
  > triggers(RESET): (y+,x-) -> z-
  > 5 transistors (3 n, 2 p)
  > Estimated delay: rising = 19.33, falling = 25.62
  > Speed independent (no timing assumptions)

SET(z')   = y x'
RESET(z') = x
[z] = z'        (output inverter)
  > triggers(SET):   (y+,x-) -> z-
  > triggers(RESET): x+ -> z+
  > 7 transistors (3 n, 4 p)
  > Estimated delay: rising = 19.33, falling = 27.25
  > Speed independent (no timing assumptions)

z = y' z + x
  > triggers(SET):   x+ -> z+
  > triggers(RESET): (y+,x-) -> z-
  > 10 transistors (5 n, 5 p)
  > Estimated delay: rising = 19.96, falling = 26.62
  > Speed independent (no timing assumptions)

z' = x' (z' + y)
[z] = z'        (output inverter)
  > triggers(SET):   (y+,x-) -> z-
  > triggers(RESET): x+ -> z+
  > 10 transistors (5 n, 5 p)
  > Estimated delay: rising = 19.58, falling = 29.71
  > Speed independent (no timing assumptions)
```

The interpretation of this information is quite obvious. Remember that the solution reported in the EQN file is only one of the possible solutions for the signal and not necessarily the best one, although petrify always tries to choose one with low cost. When generating gCs, two types of solutions may be reported in the `log` file:

- Static gates: pull-up and pull-down networks are dual and an output inverter might be required. This is the only type of solutions in the `log` file when complex gates are generated.

- Dynamic gates: a different "SET" and "RESET" network is derived. It is the designer's responsability to ensure that the gate is correctly designed to guarantee a proper voltage level when the output is not driven neither by the pull-up nor by the pull-down networks. Thus, weak feedback inverters or transistors might be required to guarantee a correct behavior of the gate.

Dynamic gates are distinguished from static gates in that two networks, namely SET and RESET, are reported in the `log` file for dynamic gates.

At this point, one might thing that everything is solved in the world of speed-independent circuits: once we have a specification of our system in terms of input/output events, we can automatically derive a speed-independent circuit, either with complex gates or generalized C elements. Nothing farthest from truth. Don't miss chapter 4.

# Chapter 4

# State encoding

State encoding is one of the most difficult problems to solve in the synthesis of asynchronous circuits. This is also one of the problems in which the capability of interaction provided by petrify can be really appreciated by the designer.

## 4.1 A simple example

Let us consider the example in Figure 4.1 that describes the behavior of the read cycle in a VME bus. The state graph depicted at the right shows two states (shadowed) with the same binary encoding, but with different future behavior. This is known as a state encoding conflict. To solve this ambiguity, some extra signals must be inserted. A state graph with no encoding conflicts is said to satisfy the *Complete State Coding* property (CSC).

The picture at the left of Figure 4.1 can be obtained with the command:

```
write_sg -bin vme_read.g | draw_astg -bin -bw -o vme_read.sg.ps
```

Petrify can automatically solve CSC as follows:

```
petrify -csc vme_read.g -o vme_read.csc
```

The result of such command is another *STG* satisfying the CSC property. A new internal signal, called `csc0`, has been inserted. Figure 4.2 shows the new *STG* and its corresponding state graph with the CSC property.

Here is where we can really appreciate the power of Petri net synthesis. After having performed transformations on the state graph (e.g. insertion of state signals), petrify is able to report the result as an *STG* again. In this way, the designer can analyze the solution and change it manually at his/her convenience.

## 4.2 How to get better solutions

Inside petrify there is a powerful engine for solving CSC that can be tuned in many different ways. Informally, the way petrify solves CSC is as follows:

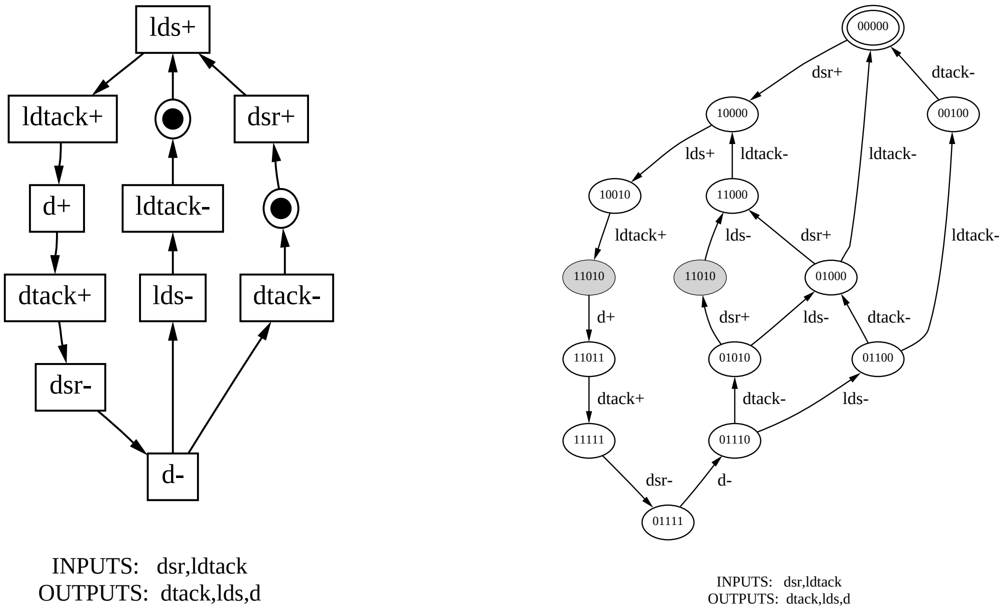



Figure 4.1: Read cycle of the VME bus: STG and state graph illustrating encoding conflicts

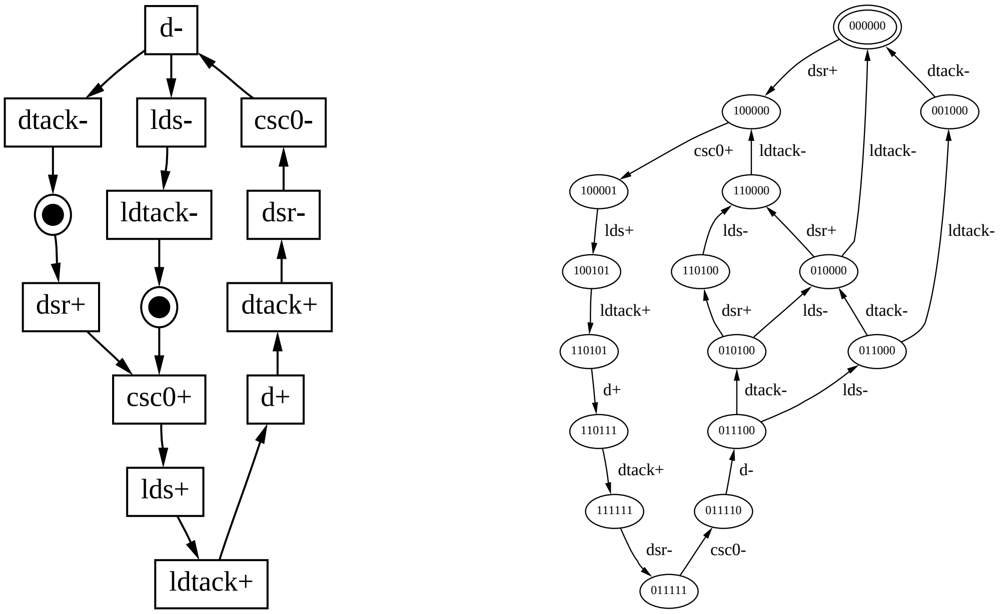



Figure 4.2: Insertion of state signal to solve CSC.

1. Find a bipartition of the state space that makes relevant progress towards solving state encoding conflicts. This bipartition is found by using heuristic search methods with some sophisticated cost function that estimates the complexity of the logic.

2. Insert a new signal according to the found bipartition

3. If not all conflicts have been solved, go to step 1

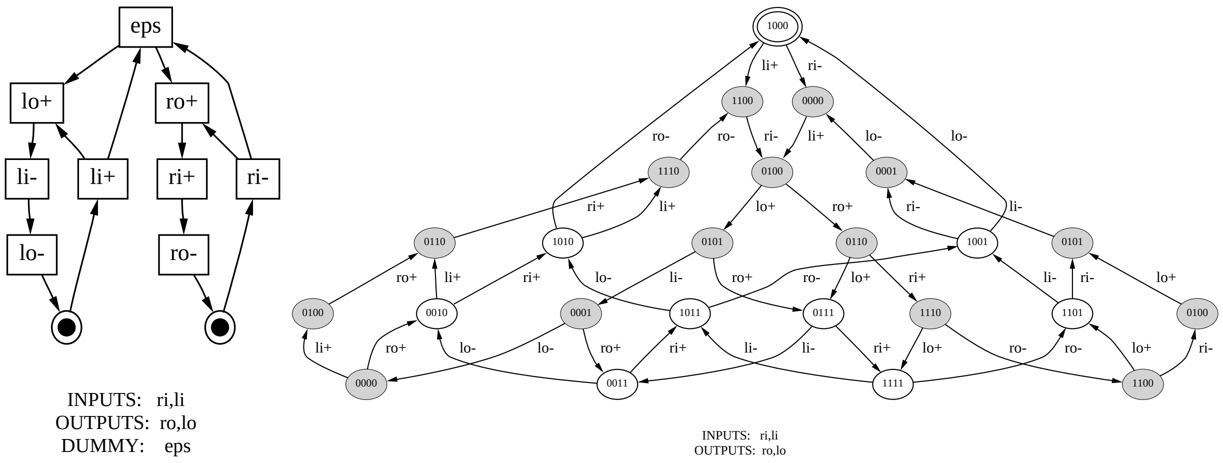



Figure 4.3: Another example with CSC conflicts (`fifo.g`)

Thus, CSC is solved stepwise by inserting a new state signal at each iteration of the previous method. However, the estimation of logic can only be accurate when the number of signals required to solve CSC is only one. In case than more than one is required, the cost function cannot find an implementation for those signals affected by the CSC conflicts. For this reason, at the earliest iterations of the algorithm, petrify inserts state signals "blindly" aiming only at reducing the number og CSC conflicts.

To alleviate this problem in those cases in which more than one CSC signal must be inserted, the following strategy can be tried by the designer:

1. Let petrify solve CSC.

2. Analyze the logic of the circuit and identify those CSC signals with higher complexity.

3. Hide one of the signals with highest complexity and solve CSC again.

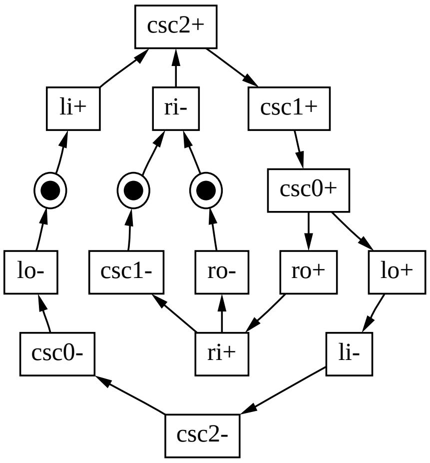
However, it is not wise to hide the latest inserted signal, since the problem left to solve will be the same as the one found by petrify before inserting that signal. Thus, petrify will report the same solution. This strategy attempts to solve CSC in a way less dependent on the insertion ordering of the signal.

Petrify has incorporated some heuristics to improve CSC in a similar way.

We will illustrate the effectiveness of this strategy with the example shown in Figure 4.3. Note that in the example there is one transition called `eps` that does not correspond to any signal event. It is just a synchronization transition between two handshakes. To obtain the state graph at the left of the figure, the transition `eps` must be first hidden since write_sg does not know how to calculate binary codes when "dummy" transitions are present. Thus, the state graph with binary encoding can be obtained as follows:

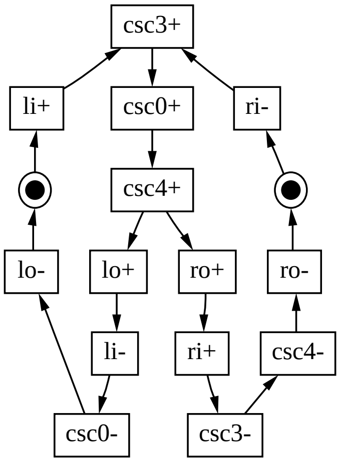
```
petrify -hide eps fifo.g | write_sg -bin | draw_astg -bin -bw -landscape -o
                                   fifo.sg.ps
```

After executing the following command




```
[ro] = ri' csc1 (csc2' + csc0);
[lo] = csc0;
[csc0] = csc1' csc0' + csc2';
[csc1] = ri' csc1 + csc0' csc2;
[csc2] = li (ri' csc1' + csc2);
```

Figure 4.4: Example `fifo.g` with CSC




```
[ro] = csc4;
[lo] = csc0 (csc4 + csc3');
[csc0] = (csc4 + csc3') (csc0' + li');
[csc3] = ri' (li csc0' + csc3);
[csc4] = csc0' csc4' + csc3';
```

Figure 4.5: Example `fifo.g` with CSC solved in an iterative way.

```
petrify fifo.g -cg -eqn fifo.eqn -o fifo.csc
```

we obtain the solution depicted in Figure 4.4. Note that the option `-csc` has not been used in this case. This is because petrify automatically invokes the CSC solving procedure when logic synthesis is required.

We now may try to solve CSC using the iterative strategy explained previously. This can be done automatically by petrify with the option -icsc.

```
petrify fifo.g -icsc -cg -eqn fifo.eqn -o fifo.csc
```

The result can now be seen in Figure 4.5. We can observe that the new logic (15 literals) is a little bit simpler than the previous one (16 literals).

To have an intuition on how petrify has solved CSC with this strategy, we can observe the on-line messages displayed during the execution:

```
  State coding conflicts for signal ro
  State coding conflicts for signal lo
The STG has no CSC.
Adding state signal: csc0
  State coding conflicts for signal lo
  State coding conflicts for signal csc0
The STG has no CSC.
Adding state signal: csc1
  State coding conflicts for signal ro
  State coding conflicts for signal csc0
The STG has no CSC.
Adding state signal: csc2
Removing signal csc1 to improve CSC iteratively.
  State coding conflicts for signal ro
  State coding conflicts for signal csc2
The STG has no CSC.
Adding state signal: csc3
Removing signal csc2 to improve CSC iteratively.
  State coding conflicts for signal ro
  State coding conflicts for signal csc0
The STG has no CSC.
Adding state signal: csc4
The STG has CSC.
```

## 4.3 Putting all the engines to work

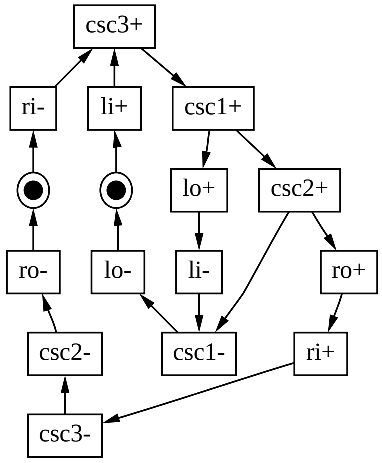
As mentioned before, petrify can be tuned in different ways to solve CSC. We now illustrate how to increase the exploration of the solution space by using two new options:

```
petrify fifo.g -icsc3 -fr10 -cg -eqn fifo.eqn -o fifo.csc
```

The result is shown in Figure 4.6. The logic complexity is now 13 literals.

The reason why the options `-icsc3 -fr10` yield a better solution can be intuitively explained.

**-icsc3:** In sevaral methods within the tool, the state space is explored in regions of states. These regions roughly correspond to "places" in Petri nets, i.e. a region is the set of states in which a place is marked. Petrify also calculates smaller blocks of states as intersections of the regions. The greater the number of intersections calculated between regions the smaller the number of states of the new generated blocks. The size of these blocks determines the granualrity at which petrify explores the space of solutions. The options `-csc` and `-icsc` can be specified with a natural number, e.g. `-csc3` or `-icsc2`. The default number is `1`. The higher the number is, the smaller the granularity of the exploration. Indeed, the smaller the granularity is, the better




```
[ro] = csc2;
[lo] = csc1;
[csc1] = (csc3' + csc2) (csc1' + li');
[csc2] = csc1' csc2' + csc3';
[csc3] = ri' (li csc1' + csc3);
```

Figure 4.6: Example `fifo.g` with CSC solved with a wider exporation of the space of solutions.

the exploration is performed, although more computationally costly. In practice, it is useless to try granularities smaller than the ones produced by `-csc3`. The default value often yields similar solutions.

**-fr10:** The exploration of the partitions of the state space for state signal insertion is done in a similar way as many computer game programs play. Think about programs playing chess. The program analyzes a position and starts calculating all possible combinations starting from the current position, thus generating a search tree. At each level of this tree, the program heuristically selects a subset of promising solutions that will be the seeds for the next level of the tree. This subset of solutions selected at each level is what petrify calls the "frontier" of solutions. The option `-fr` defines the size of this frontier. Indeed, the wider the frontier, the larger the set of solutions explored at the expense of computational cost. If you are not in a hurry in finding a solution and the problem to solve is not too large, you might even try the option `-fr100`. If not specified, petrify works as follows: in the first levels of the tree the frontier is around 10, but decreases at each iteration until it reaches the minimum value of 1. When `-fr`$n$ is specified, the value of the frontier remains constant during the exploration.

## 4.4 When CSC cannot be solved

Unfortunately, petrify cannot guarantee a solution to the state encoding problem even if it exists. Whenever that occurs, the designer must take some action in order to find a solution or make the task easier for petrify. Here are some suggestions:

- Try to make the exploration of the solution space more aggressive by using the options `-csc`$n$ and `-fr`$w$, as explained in the previous section.
- Change the specification. A possibility here is to use the option `-er` (see section 2.3) to split events with disconnected excitation regions.

- Define some inputs events as "slow" events to increase the chances of finding places to insert state signals (see section 4.5 for more details).
- Make timing assumptions (see chapter 5).

However, there are conflicts for which no solution exists unless the specification is changed or timing assuptions are made in such a way that a non-speed-independent circuit can be derived. These are known as irreducible conflicts. Next section will discuss how to solve them.

## 4.5 Irreducible CSC conflicts

Let us look at the example in Figure 4.7. After executing the command

`petrify -csc abc.g -o abc.csc`

we observe the following messages during execution:

```
  State coding conflicts for signal c
The STG has no CSC.
Adding state signal: csc0
  State coding conflicts for signal csc0
The STG has no CSC.
Adding state signal: csc1
Warning: irreducible CSC conflicts found.
See the log file for details about traces.
>>> ERROR: Cannot solve CSC.
```

The resulting specification (Petri net and state graph) is depicted in Figure 4.8. Let us now look at the `log` file (`petrify.log`). We can read the following messages:

```
Error: CSC cannot be solved.
Irreducible conflicts found.
  Trace of events: a+ csc0+ c+ [ b+ a- b- a+ ]
CSC cannot be solved unless:
  - The I/O interface is changed or
  - Relative timing assumptions are specified
```

The reported trace of events indicates how the irreducible CSC conflict can be reached. After some initialization trace (`a+ csc0+ c+`) a state with CSC conflict is reached (indicated by the symbol `[`). Then a trace of input events is executed until another state with the same binary encoding is reached. Note that the trace of input events (`[ b+ a- b- a+ ]`) is always a complementary set of signal transitions, i.e. each signal appearing in the trace has an even number of events. To disambiguate the states at the beginning and end of the input trace, some internal signal must be inserted in the middle of the trace. This is unacceptable for an speed-independent circuit.

The only way to overcome this problem is to "tolerate" the insertion under one of the following assumptions:

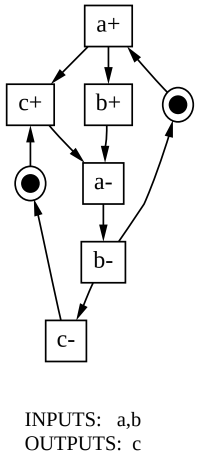



Figure 4.7: Example `abc.g` with irreducible CSC conflicts.

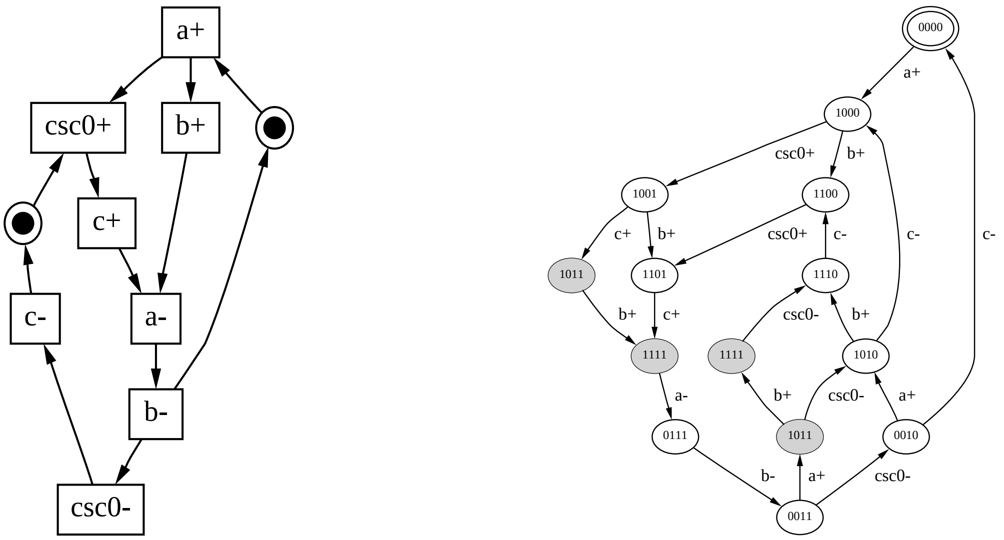


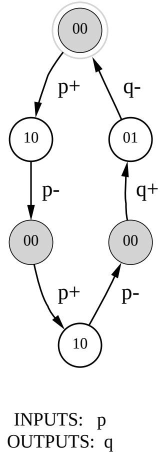

Figure 4.8: Example `abc.g` after trying to solve CSC conflicts.

- Assume that the environment will have to consider the inserted signal as observable (even we call them "internal") and will not react until the new signal has stabilized when excited.
- Assume that the environment will be slow enough to allow the internal signal stabilized when excited (thus violating the assumption of speed-independence).

Petrify enables the designer to declare some input events as "slow". We will illustrate how this can be useful in the following example.

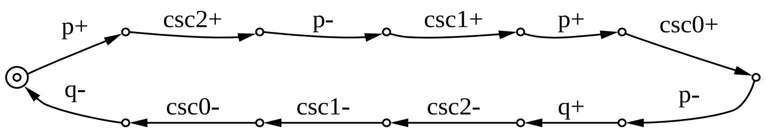
### 4.5.1 Another example with irreducible CSC conflicts

The state graph in Figure 4.9 specifies the behavior of a modulo-2 counter. The circuit generates one pulse at the output `q` every two pulses observed at the input `p`. In this case




```
.inputs p
.outputs q
.state graph
s0 p+ s1 p- s2 p+ s3 p- s4 q+ s5 q- s0
.marking {s0}
.slowenv
.end
```

Figure 4.9: Example `counter.g` with irreducible CSC conflicts and slow events.




```
[q] = p' csc0;
[csc0] = p' csc0' + csc1';
[csc1] = csc2 (p' + csc1);
[csc2] = q' csc2 + p;
```

Figure 4.10: Solution for `counter.g`.

the format to describe state graphs in petrify has been used.

This specification has irreducible CSC conflicts. If we start fom the initial state and after firing the trace `p+ p- p+ p-` we reached another state with the same encoding as the initial state, but with different behavior for the output signal `q`.

However, we can assume that the pulses produced by the environment are wide enough that any internal activity inside the circuit will be allowed to stabilize. This can be indicated by the statement `.slowenv` that indicates that **all events of the input signals are slow**. In the case that assumption should be made for only a subset of the input events, the statement `.slow` can be used, e.g. `.slow p-`. When the `astg` format is used to specify *STG*s, the `.slow` statement can be followed by individual instances of signal transitions, e.g. `.slow a+/1 a+/2 b- c-/4`.

Let us now solve CSC and derive logic equations.

```
petrify counter.g -cg -eqn counter.eqn -o counter.csc
```

The obtained solution is depicted in Figure 4.10

## 4.6 Timing constraints imposed by internal activity

If we look at the solution presented in Figure 4.10, we can observe that some of the input events are “delayed” by non-observable internal events. Obviously, this violates the input/output interface initially specified for the circuit. However, we have also seen that this violations are tolerated under some timing assumptions.

Whenever some of these timing assumptions are efectively used to make some transformations of the state space, `petrify` gives some warning messages in the `log` file. The first part of `petrify.log` after having solved CSC is the following:

```
Number of added state signals : 3
----------------------------------------------------
| Paths of internal events delaying input events: |
----------------------------------------------------
    csc2+ => p-/1
    csc1+ => p+/2
    csc0+ => p-/3
```

The information concerning "delaying input events" indicates that the new specification involves some internal activity that must be allowed to stabilize before new input events occur. As an example, the first message indicates that event `csc2+` is "in progress" when `p-/1` is already enabled. For a safe functioning of the circuit, `csc2+` should occur before `p-/1` occurs. These are timing assumptions that should be formally verified or validated within the actual environment in which the circuit is embedded.

The second section of the file concerns "Output $\rightarrow$ Input Delays":

```
----------------------------
| Input -> Input Delays: |
----------------------------
Average delay =  1.25 events
Worst-case delay =  5.00 events
Input events with worst-case delay: p+
  > Input events preceding p+: p-/3(5)
  > Input events preceding p-/1: p+(0)
  > Input events preceding p+/2: p-/1(0)
  > Input events preceding p-/3: p+/2(0)
```

This is an attempt to provide information about response time of the circuit. It roughly estimates the number of non-input events that must be fired before an input event is enabled. In the solution shown in Figure 4.10 we can observe that there is one input event with five non-input transitions preceding it:

$$(q+ \rightarrow csc2- \rightarrow csc1- \rightarrow csc0- \rightarrow q-) \longrightarrow p+$$

This is the critical response time of the circuit. In all other cases, the circuit only fires one event before enabling some input event of the environment. In average, the response time is 1.25 events.

# Chapter 5

# Synthesis with relative timing assumptions

The synthesis of speed-independent circuits assumes a delay model that can be considered too conservative for the temporal behavior we may expect from the actual environment of the circuit and the technology used to implement it.

By making some timing assumptions on the behavior of the environment and the circuit itself, the logic complexity of the circuit can be simplified. However, the property of speed-independence may be lost, i.e. the circuit may not react correctly for any delay of the components of the system. For this reason, it is crucial to know under which conditions the circuit behaves properly.

Petrify incorporates synthesis methods for hazard-free circuits under timing assumptions. It also provides backannotation that indicates the required timing constraints for a proper functioning of the circuit.

## 5.1 Timing assumptions

In this section, the timing assumptions that petrify can "understand" will be discussed. All of them are *relative timing assumptions* which means that they refer to the specific ordering of events with regard to other events. In contrast, absolute timing assumptions are specified in terms of time intervals for the occurrence or enabling of events.

The difference between relative and absolute timing assumptions is illustrated in Figure 5.1. The figure depicts a part of a Petri net (arcs implicitly represet 1-input 1-output places that can hold tokens). Assume that the designer knows the execution delays for some of the events and these delays can be expressed as time intervals, e.g. $\delta(b) = [1,3]$, $\delta(c) = [2,4]$, $\delta(d) = [6,8]$ and $\delta(e) = [2,3]$. The delays for events $a$ and $x$ are unknown.

Under the previous assumptions, it can be deduce that $b$ will always occur before $e$, but no assumption can be made about the ordering of events $c$ and $d$, since the enabling time of event $c$ is also determined by the firing time of event $x$. These analysis would roughly correspond to the analysis done when absolute timing assumptions are used.

With relative timing assumptions the designer can direcly indicate that $b$ will occur before $e$ and that $c$ will occur before $d$. Indeed, it is the designer's responsability to ensure that

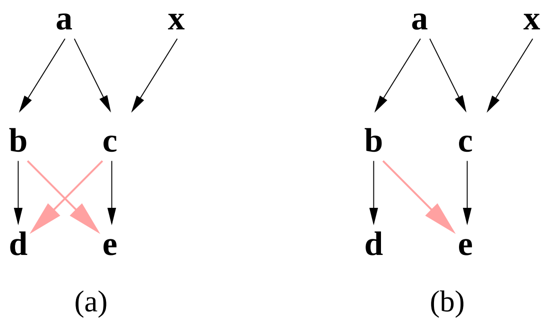



Figure 5.1: Relative and absolute timing assumptions.

those assumptions hold. The fact that $c$ will occur before $d$ can be deduced by the designer if it is known, for example, that $x$ will always occur before $a$.

Three different types of relative timing assumptions can be specified in `petrify`:

- Firing order of concurrent events
- Simultaneous occurrence of concurrent events
- Early enabling

To properly understand the semantics of relative timing constraints, the concepts of *enabling region* and *firing region* must be introduced. Given a state graph, the *enabling region* of event $a$, $EN(a)$ is defined as the set of states in which $a$ is enabled. The *firing region* of event $a$, $FR(a)$, is defined as the set of states in which $a$ is allowed to fire.

Whereas in speed-independent circuits, both concepts are the same (an event can fire as soon as it is enabled), they substantially differ when timing assumptions are considered. An event can be enabled at some state but cannot fire until the system reaches its firing region.

All these timing assumptions are specified in the input file where the $STG$ is described. They should go after the graph (or state graph) specification and before the `.end` statement.

### 5.1.1 Firing order of concurrent events

This timing assumption is specified with the following syntax:

$$.\text{time a} <| \text{ b}$$

The meaning of that statement is the following:

> *Whenever events a and b are concurrent (i.e. simultaneously enabled and not in conflict), a will always fire before b*

To illustrate how the assumption is made by `petrify` we will use the example of Figure 5.2.

We can observe that events $a$ and $b$ have different relations. They can be concurrent (both enabled in state $s_0$) or ordered ($a$ is enabled in $s_2$ and $b$ is enabled in $s_7$). The timing assumption only applies to those states in which the events are concurrent.

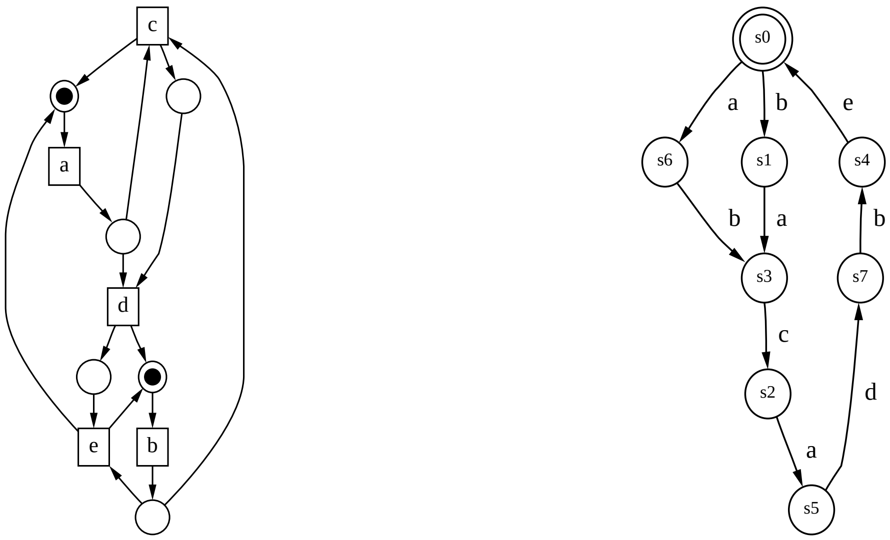



Figure 5.2: Illustration of timing assumptions on event ordering.

The application of the timing assumption would mean that $b$ will not fire in $s_0$. As a consequence, state $s_1$ will become unreachable in the timed domain.

With regard to event $b$, petrify also considers that $s_0$ is a "don't care" state for the enabledness of event $b$, i.e. after logic synthesis two different solutions could be reported:

- One in which $b$ is enabled in $s_0$. Still, the timing assumptions will make $s_1$ unreachable, thus $b$ will not fire until $s_6$ is reached.
- One in which $b$ is not enabled in $s_0$. In this way, the ordering $a \to b$ will be forced by the logic of the circuit, i.e. no timing assumption is required for this solution to be valid.

This can be formally expressed as follows:

$$\{s_6\} = FR(b) \subseteq EN(b) \subseteq \{s_0, s_6\}$$

Petrify will choose a solution for $EN(b)$ that minimizes the cost of the logic.

### 5.1.2 Simultaneous occurrence of concurrent events

This timing assumption is specified with the following syntax:

.time a=b@c

An example of this assumption is shown in Figure 5.3. The meaning of the simultaneity timing assumption is the following:

> *Let us take the states in which a and b are enabled and concurrent. Event c will not fire in any of the successor states until a and b have fired. This assumption only applies when c is triggered by either a or b or both.*

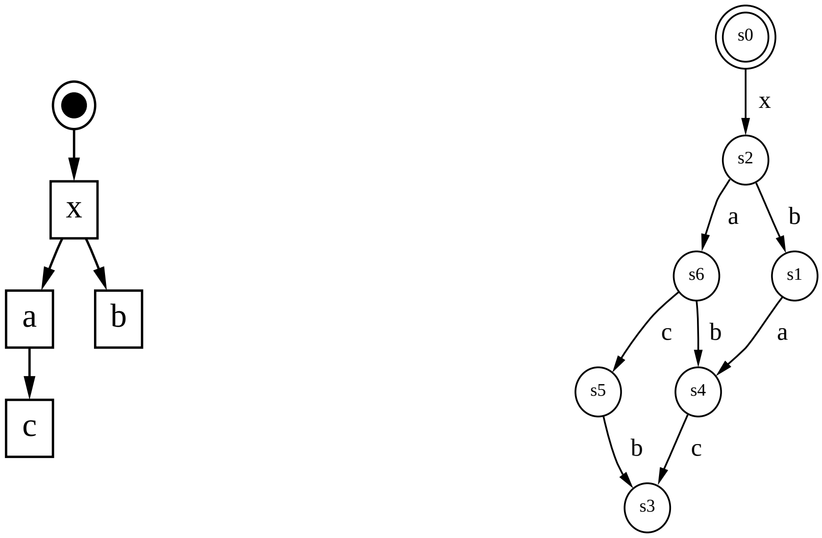



Figure 5.3: Simultaneity timing assumptions

Informally, the assumptions describes the situation in which the firing time difference between $a$ and $b$ is not distinguishable by event $c$. Looking a the example, this would mean that the system would produce the same observable behavior if $c$ would be "triggered" by $b$ or by both events $a$ and $b$.

Looking at the state graph of Figure 5.3, the simultaneity constraint indicates that $c$ would not fire in state $s_6$ and, therefore, state $s_5$ would become unreachable. On the other hand, event $c$ would be allowed to be enabled in the states $s_1$, $s_4$ and $s_6$. Thus,

$$\{s_4\} = FR(c) \subseteq EN(c) \subseteq \{s_1, s_4, s_6\}$$

The simultaneity timing assumptions can be extended to larger number of events as follows:

```
.time a=b=c=d@x,y,z
```

meaning that the firing time of events $a$, $b$, $c$ and $d$ is considered to be non-distinguishable with respect to events $x$, $y$ and $z$.

### 5.1.3 Early enabling

This timing assumption will be illustrated by the example in Figure 5.4.

Early enabling assumptions are specified with the following syntax:

```
.time c>b
```

The formal meaning of this assumption is the following:

> *Let us take all those states $s - i$ in which $c$ is not enabled, but becomes enabled after firing $b$ (i.e. $b$ triggers $c$). Then, the states $s_i$ can potentially belong to* EN$(c)$.

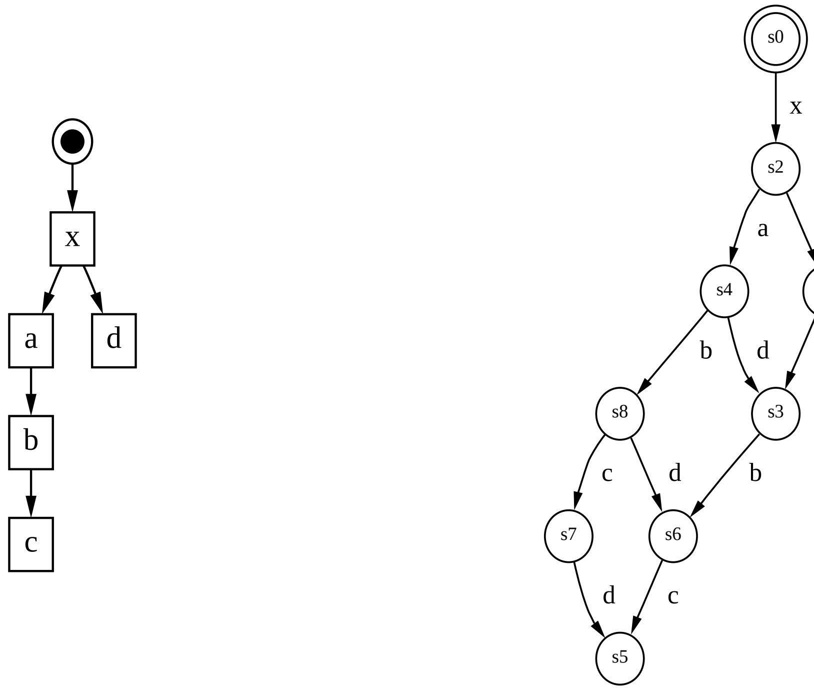



Figure 5.4: Example to illustrate early enabling assumptions.

Informally, this assumptions indicates that event $c$ can be enabled before it must fire. However, the delay of the logic implementing $c$ will ensure that $c$ will fire after $b$. Indeed, this may be seen as a risky assumption, since the logic of signals $b$ and $c$ are not known before logic synthesis. Relative timing assumptions require a post-verification than ensure their validity. In the case they do not hold, some actions must be taken, for example:

- Resynthesize the circuit without those invalid assumptions or
- Change the delays of the components of the circuit (e.g. by transistor sizing or delay padding) so that the assumptions become valid.

In the previous example, $EN(c)$ and $FR(c)$ are defined as follows:

$$\{s_6, s_8\} = FR(c) \subseteq EN(c) \subseteq \{s_3, s_4, s_6, s_8\}$$

The early enabling assumptions can be extended to chains of events. In the previous example, the following assumptions could be also specified:

`c>b>a`

indicating that $EN(c)$ can be extended up to the enabling of event $a$. However, no assumption is made about the enabledness of $b$ with regard to $a$.

## 5.2 The $xyz$ example

Let us illustrate the usage of timing assumptions in the well-known $xyz$ example shown in Figure 5.5.

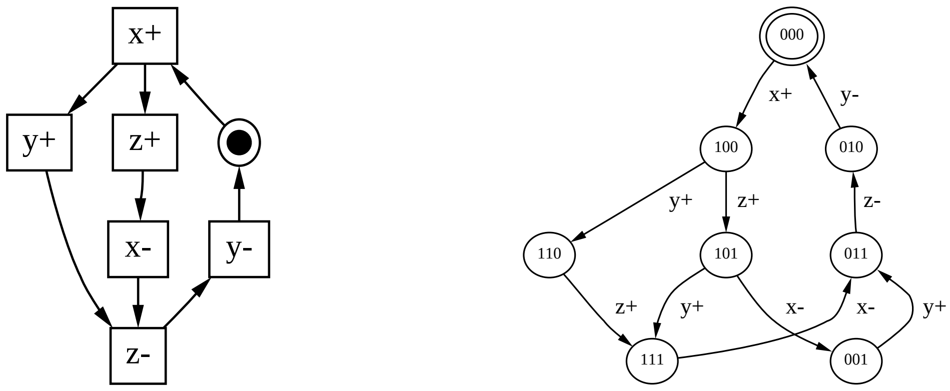



Figure 5.5: The $xyz$ example.

A speed-independent implementation of the behavior with complex gates would be the following:

```
[x] = z' (x + y');
[y] = z + x;
[z] = y' z + x;
```

Let us assume that $x$, $y$ and $z$ are output signals and we conjecture that the firing of $y+$ will occur always before the firing of $x-$. We can intuitively deduce this by observing that $y+$ is always enabled before $x-$ and that the logic for signal $x$ in the speed-independent implementation looks more complex than the logic for signal $y$.

We can then add some timing assumption to our specification:

```
.outputs x y z
.graph
x+ y+ z+
z+ x-
y+ z-
x- z-
z- y-
y- x+
.marking {<y-,x+>}
.time y+<|x-
.end
```

and then synthesize a circuit with the following command:

```
petrify xyz.g -cg -topt -eqn xyz.eqn -no
```

The option `-topt` indicates that `petrify` must take timing assumptions into consideration to derive logic. The following logic is obtained:

```
[x] = z' (x + y');
[y] = z + x;
[z] = x;
```

in which we observe a drastic simplification of the logic for signal $z$. Petrify has taken advantage of the timing assumption to consider the state 001 as unreachable and implement $z$ simply as a buffer.

But as important as the solution is the feedback that petrify reports about the timing assumptions used for each solution. If we look at the file `petrify.log`, we can analyze different solutions for each signal. In particular, among the solutions reported for signal $z$, we have the following:

```
z = x
  > triggers(SET):      x+ -> z+
  > triggers(RESET):    x- -> z-
  > 4 transistors (2 n, 2 p)
  > Estimated delay: rising = 19.33, falling = 16.00
  > Timing assumptions (concurrency):  y+<x-

[... other solutions ...]

z = y' z + x
  > triggers(SET):      x+ -> z+
  > triggers(RESET):    (y+,x-) -> z-
  > 10 transistors (5 n, 5 p)
  > Estimated delay: rising = 19.96, falling = 26.62
  > Speed independent (no timing assumptions)
```

The first solution corresponds to the one obtained with timing assumptions. The second one corresponds to the speed-independent implementation. There are two important informations reported for the solutions:

**Timing assumptions.** Petrify indicates that the solution `z=x` is only valid under the assumption that $y+$ fires before $x-$. The solution `z = y' z + x` is valid under any timing asumption (speed-independent solution). The timing assumptions reported by petrify are not necessarily the same as the one defined in the specification. Petrify always tries to report the less stringent assumptions that make the solution valid.

**Trigger signals.** For each solution petrify also indicates which events are triggered the rising and falling transitions of the signal. Note the difference for the trigger events of $z-$. In the "timed" solution, $y+$ is no longer triggering $z-$ since it is assumed to fire before $x-$. This information is much more relevant when "early enabling" assumptions are done for synthesis.

Let us now try another type of timing assumption. We can also observe that $y+$ and $z+$ are enabled simultaneously. If the delays of their gates would be similar, we could consider that the firing time of $y+$ and $z+$ would not be distinguishable with respect to event $x-$. The specification now would be as follows:

```
.outputs x y z
.graph
x+ y+ z+
z+ x-
y+ z-
x- z-
z- y-
y- x+
.marking {<y-,x+>}
.time y+=z+ @ x-
.end
```

After executing the same command as above, the following solution is obtained:

```
[x] = y';
[y] = z + x;
[z] = x;
```

This is a significant improvement ! but ... under which assumptions is this solution correct? Again, the file `petrify.log` will give us relevant information about that. Let us first look at different solutions for $x$ and compare with the speed-independent solution:

```
x = y'
  > triggers(SET):     y- -> x+
  > triggers(RESET):   y+ -> x-
  > 2 transistors (1 n, 1 p)
  > Estimated delay: rising = 15.21, falling = 9.12
  > Concurrency reduction:  y+==>x-
  > Timing assumptions (early enabling):  z+<x-

x = y' z'
  > triggers(SET):     y- -> x+
  > triggers(RESET):   (y+,z+) -> x-
  > 4 transistors (2 n, 2 p)
  > Estimated delay: rising = 31.25, falling = 9.38
  > Timing assumptions (early enabling):  z+<x-

[... other solutions ...]

x = z' (x + y')
  > triggers(SET):     y- -> x+
  > triggers(RESET):   z+ -> x-
  > 8 transistors (4 n, 4 p)
  > Estimated delay: rising = 33.12, falling = 9.38
  > Speed independent (no timing assumptions)
```

The solution `x = y'` is generated by disabling $x-$ in state 101 (where it was initially enabled). This makes the state 001 unreachable. But note that, in this case, it is not unreachable due to timing assumptions, but due to the fact that the logic for $x$ does not enable $x-$ in state 101. This is what the message

```
> Concurrency reduction:  y+==>x+
```

means. In other words, the fact that $y+$ fires before $x-$ fires does not need to be verified by making timing assumptions. It is something that the logic of the circuit already guarantees.

Still there is another inmportant assumption for that solution:

```
> Timing assumptions (early enabling):  z+<x-
```

This indicates that $x-$ is enabled in such a way that it becomes concurrent with $z+$ (now $x-$ is also enabled in state 110). The assumption for correctness is that $z+$ should fire before $x-$. To verify that assumption at circuit level, one might need some additional information: how much early is $x-$ enabled? Again, this information is provided in the section of trigger signals. We may realize that now it is $y+$ that triggers $x-$ (it was $z+$ in the speed-independent solution). Thus, by combining these informations we can determine when the enabling of an event is started (trigger events) and the event is allowed to fire (when other concurrent events have already fired). Now, it is time for the designer to decide whether these assumptions are realistic or can be met by the implementation.

Still, there is another interesting solutions that appears in the file `petrify.log`: `x = y' z'`. This solution makes the state 001 again reachable, since $x-$ is enabled in state 101. But $x-$ is also enabled earlier in state 110 and the assumption `z+<x-` must still be ensured.

An important aspect of the information provided by petrify is that different timing assumptions must be ensured for each different solution of each signal. The selection of one solution for one signal does not affect the assumptions made for other gates. However, the combination of all the solutions for each gate may lead to a set of simpler constraints. Currently, it is the designer who has to figure out how these constraints interact among them. What petrify guarantees is that the solutions will be valid if the timing assumptions reported for each individual solution of each signal are met.

## 5.3 Automatic derivation of timing assumptions

Bearing in mind that the designer tipically likes to improve the performance of the circuits and that some assumptions are easily derivable by just looking at the structure of the specification, petrify has also made an effort to automatically generate "reasonable" timing assumptions.

With this strategy, petrify only wants to simplify the job of the designer, but not to susbtitute it. Indeed, petrify will finally report which assumptions have been generated automatically and which of them are actually used for each solution. It is then the designer's Indeed, petrify will finally report which assumptions have been generated automatically and which of them are actually used for each solution. It is then the designer's responsability to guarantee tha validity of the timing assumptions.

### 5.3.1 Delay model for the automatic derivation of timing assumptions

The following model is considered for the delay of a signal transition (delay from its enabling time to its firing time):

**Non-input signals:** each gate implementing a non-input signal has a delay in the interval $[1-\varepsilon, 1+\varepsilon]$, where $\varepsilon < 1/3$. Thus, the delay of two gates is always greater than the delay of one gate.

**Input signals:** have a delay in the interval $(1+\varepsilon, \infty)$. Thus, the delay of the environment is always greater than the delay of one gate.

**Slow input signals:** have a delay in the interval $[k, \infty)$, where $k$ any arbitrarily large constant. The delay $k$ indicates that the enabling of a slow input signal transition always allows the completion of any internal activity in the circuit (firing of enabled non-input signals).

**Delay padding:** the delay of any gate implementing any non-input signal can be lengthened after logic synthesis, e.g. by transistor sizing or delay padding, to meet the required timing assumptions.

The automatic generation of timing assumptions by using this delay model will be illustrated with the example of Figure 5.6. $i_1 \ldots i_4$ are input signals, whereas the rest are output signals. Moreover, the event $i_4+$ is declared to be "slow" by using the statement[1]:

```
.slow i4+
```

#### Ordering of concurrent events

The assumptions made on the relative order of two concurrent events, $a$ and $b$ are the following:

- If $a$ is always enabled before $b$ and $a$ is not an input event, then $a$ is assumed to fire before (`.time a<|b`). In the example of Figure 5.6 this assumption would apply, among others, to the pairs of events $a+ \rightarrow i_4+$ and $b+ \rightarrow g+$.

- If $a$ and $b$ are simultaneously enabled, $a$ is a non-input event and $b$ is an input event, then $a$ is assumed to fire before (`.time a<|b`). There is no such case in the example of Figure 5.6, but it would correspond to the pair of events $a+ \rightarrow b+$ if $b$ would be an input event.

- If $a$ and $b$ are simultaneously enabled and both are non-input events, petrify selects heuristically an order between both (tipically this order determines that the event with simpler logic will fire first, although it may not be necessarily true when the actual gates are derived after logic synthesis).

[1] see Chapter 4 for further information about slow input events

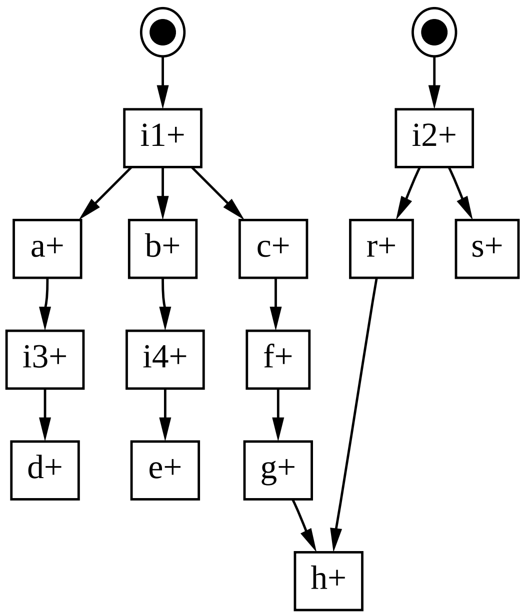



Figure 5.6: Example for automatic generation of timing assumptions.

- No assumptions are done for pairs of events that can be enabled in different order in the untimed domain. For example, for events $i_4+$ and $f+$ we can find event traces in which $i_4+$ is first enabled (e.g. $i_1+, b+, c+$) and in which $f+$ is first enabled (e.g. $i_1+, c+, b+$).

### Simultaneous trigger events

In case two non-input events, $a$ and $b$, are enabled simultaneously and another event $c$ is triggered by $a$ or $b$ (or both), a simultaneity assumption is automatically generated (`a=b@c`).

Unfortunately, petrify only makes this analysis for pairs of simultaneous events. Assumptions on more than two simultaneous events are left to be specified by the designer.

There are several examples of simultaneity assumptions in Figure 5.6: `b+=c+@f+`, `r+=s+@h+`, etc.

### Early enabling

When several non-input events have a trigger relation among them, petrify automatically generates *early enabling* assumptions, taking into account that the delays of the gates can be properly lengthened to meet the ordering relations of the specificatin.

In the example of Figure 5.6, there are chains of events that have a trigger relation among them: $c+ \rightarrow f+ \rightarrow g+ \rightarrow h+$ and $r+ \rightarrow h+$. The following assumptions are automatically generated:

```
.time f+ > c+
.time g+ > f+ > c+
.time h+ > g+ > f+ > c+
.time h+ > r+
```

**Ordering of slow input events: a generalization of the fundamental mode assumption**

Under the assumption that the delay of slow input events is long enough to enable the circuit stabilize when other internal activity is in process, petrify can automatically generate assumptions on the firing order of slow events.

The timing analysis is performed in such a way that only concurrent non-input events with a common predecessor history with the slow input event are assumed to fire first. This intuitive idea will be more clear after looking at the example in which $i_4+$ is the only slow input event.

Events $i_4+$ and $g+$ have a common predecessor event in their history: $i_1+$. Moreover, no other input events precede the enabledness of $i_4+$ and $g+$ since the firing of $i_1+$. If we consider the firing time of $i_1+$ as the starting point for timing analysis ($t = 0$), and considering the delay model for automatic assumptions, $g+$ will fire in the interval $[3(1-\varepsilon), 3(1+\varepsilon)]$, i.e. three gate delays. On the other hand, the firing interval for $i_4+$ will be in the interval $[k+1-\varepsilon, \infty)$, where $k$ can be arbitrarily large. Therefore, petrify will deduce that the firing time of $g+$ will be always before the one for $i_4+$.

Note that this assumption does not hold when the considered event is $h+$, since no common preceding history with $i_4+$ can be found with no other input events enabled in between. For a similar reason, no ordering assumption can be made for the firing of $i_4+$ and $d+$ since an input event ($i_3+$) precedes $d+$ before meeting the common preceding point in which timing analysis can start.

Technically, this common point in the history of two events is called *local nodal point*. The analysis based on local nodal points generalizes the concept of fundamental node typically used for the synthesis of *burst-mode* specifications. Burst-mode machines work under the assumption that each state of the specification is a global nodal point, i.e. no non-input activity is enabled in the state. From the point of view of specification, fundamental mode does not allow any concurrency between the environment and the circuit.

The notion of *slow input event* takes the advantages of fundamental mode assumptions (logic can be simplified) and speed-independent assumptions (concurrency is not sacrifized). As an example, the definition of slow input events would allow to synthesize a system with several sets of handshake signals by assuming a "local" fundamental mode operation with each individual handshake, but maintaining the concurrency among different independent sets of handshakes. One of these examples is the VME bus controller described in Chapter 6.

Finally, all the automatically generated assumptions for the example of Figure 5.6 are the following:

```
.time b+<|a+
.time c+<|a+
.time a+<|i4+
.time a+<|e+
.time a+<|f+
.time a+<|g+
.time a+<|h+
.time c+<|b+
```

```
.time b+<|i3+
.time b+<|d+
.time b+<|f+
.time b+<|g+
.time b+<|h+
.time c+<|i3+
.time c+<|d+
.time c+<|i4+
.time c+<|e+
.time s+<|h+
.time s+<|r+
.time a+<|i4+
.time c+<|i4+
.time f+<|i4+
.time g+<|i4+
.time f+>c+
.time g+>f+>c+
.time h+>g+>f+>c+
.time h+>r+
.time a+=c+@f+
.time b+=c+@f+
.time r+=s+@h+
```

## 5.4 The $xyz$ example revisited

Even though the $xyz$ example is simple, still significant improvements in logic can be obtained by applying automatic timing assumptions on it. Chapter 6 will focus on a more complex example to illustrate all the features of petrify on synthesis with and without timing assumptions.

Let us take the $xyz$ example shown in Figure 5.5 without any timing assumption and execute the command:

```
petrify xyz.g -cg -atopt -eqn xyz.eqn -no
```

The option `-atopt` indicates that petrify must generate automatic timing assumptions. In case the designer had already specified some assumptions, the new ones would be added to the designer's ones. In any case, petrify takes care that the assumptions automatically generated are not contradictory with the ones specified by the designer.

The resulting circuit is shown in Figure 5.7. Still, this is the solution reported in the file `xyz.eqn`, but is it a valid solution? Can the timing assumptions for such solution be met?

Let us look at the file `petrify.log`. We can observe that the following assumptions have been automatically generated:

```
.time z+<|y+            # concurrency reduction (automatic & simultaneous)
.time y+<|x-            # concurrency reduction (automatic)
```

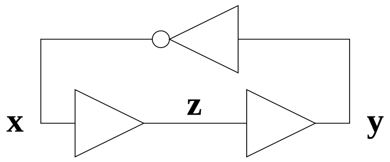



Figure 5.7: Optimized circuit for the $xyz$ example.

```
.time x+>y->z->y+       # early enabling (automatic)
.time x+>y->z-          # early enabling (automatic)
.time x->z+             # early enabling (automatic)
.time z->y+             # early enabling (automatic)
.time z->x-             # early enabling (automatic)
.time y->z-             # early enabling (automatic)
.time y->z->x->z+       # early enabling (automatic)
.time y+=z+@x-          # simultaneity (automatic)
```

and the following information is reported for the solutions:

```
x = y'
  > triggers(SET):     y- -> x+
  > triggers(RESET):   y+ -> x-
  > 2 transistors (1 n, 1 p)
  > Estimated delay: rising = 15.21, falling = 9.12
  > Concurrency reduction:  y+==>x-
  > Timing assumptions (concurrency):  z+<y+

...

y = z
  > triggers(SET):     z+ -> y+
  > triggers(RESET):   z- -> y-
  > 4 transistors (2 n, 2 p)
  > Estimated delay: rising = 19.33, falling = 16.00
  > Concurrency reduction:  z+==>y+
  > Speed independent (no timing assumptions)

...

z = x
  > triggers(SET):     x+ -> z+
  > triggers(RESET):   x- -> z-
  > 4 transistors (2 n, 2 p)
  > Estimated delay: rising = 19.33, falling = 16.00
  > Timing assumptions (concurrency):  y+<x-
```

The provided timing information can be summarized as follows:

- The firing order $z+ \rightarrow y+$ is required for the solution `x = y'` to be valid, but the firing order $z+ \rightarrow y+$ is enforced by the solution `y = z` that reduces concurrency and makes the state 110 unreachable.

- The firing order $y+ \rightarrow x-$ is required for the solution `z = x` to be valid, but the firing order $y+ \rightarrow x-$ is enforced by the solution `x = y'` that reduces concurrency and makes the state 001 unreachable.

Therefore, the enforced concurrency reduction ensures the validity of the timing assumptions and a speed-independent circuit is obtained ! This is an example on how concurrency reduction does not always imply a loss of performance. The reduction on logic results in a more efficient circuit.

# Chapter 6

# The synthesis of a VME bus controller

This chapter illustrates the use of petrify for the synthesis of a VME bus controller. Special emphasis will be done on the synthesis methodology with relative timing assumptions.

## 6.1 The VME bus controller

Figure 6.1 shows the I/O interface of a VME bus controller that controls the communication of a device with the bus through a data transceiver (signal $D$). At the side of the device there is a pair of handshake signals that follow a four-phase protocol ($LDS$ and $LDTACK$). At the side of the bus there are two input signals ($DSr$ and $DSw$) that follow a four-phase protocol with the output signal $DTACK$. $DSr$ and $DSw$ indicate the beginning of a READ and WRITE cycle respectively. The timing diagram corresponding to a READ cycle is depicted in Figure 6.1(b).

The *STG*s corresponding to the READ and WRITE cycles are shown in Figures 6.2(a) and 6.2(b) respectively.

For the synthesis of the VME bus controller, we need to derive a specification that includes the behavior of the READ and WRITE cycles. This can be done by using choice places in our Petri net formalism that model the nondeterminism of the environment. In this

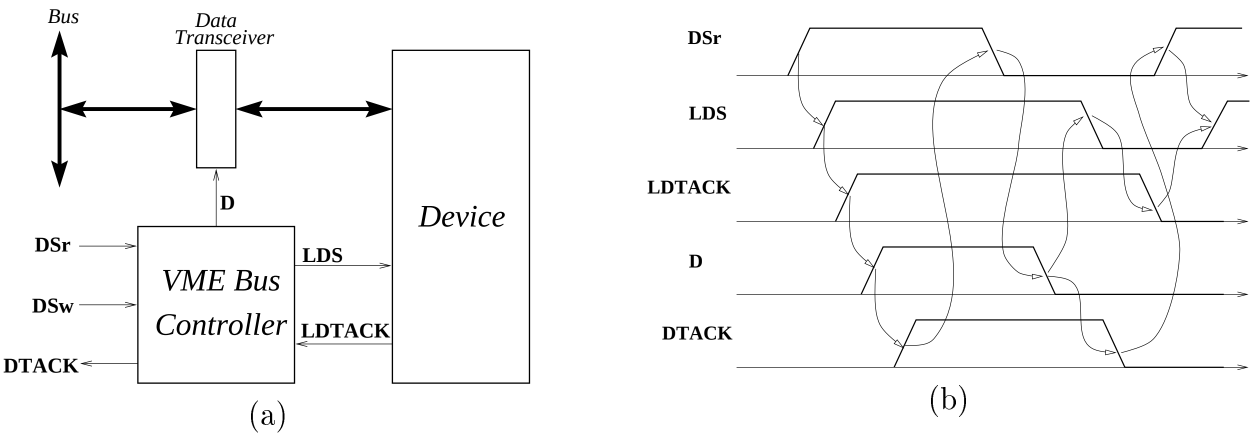
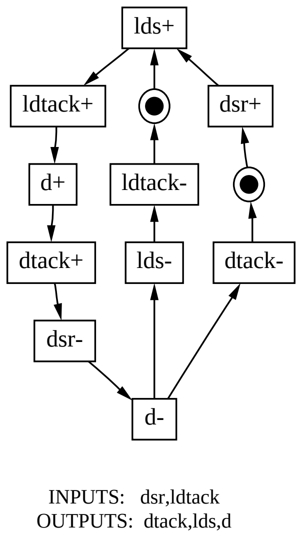
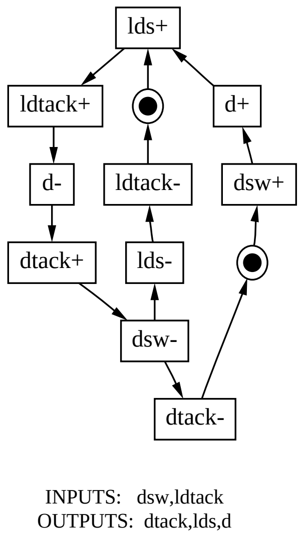



Figure 6.1: (a) VME bus controller. (b) READ cycle.




(a)




(b)

Figure 6.2: *STG*s for the (a) READ and (b) WRITE cycles.

case, the nondeterminism comes from the fact that the environment can choose to initiate a READ or a WRITE cycle after the completion of the previous cycle.

An *STG* describing the complete behavior of the controller is shown in Figure 6.3(a). We can observe that some signal transitions, e.g. $lds+$ have multiple instances in the *STG*. The indices `1` and `2` have been used to distinguish events in the READ and WRITE cycles respectively.

There is also the possibility of representing the whole behavior in an *STG* with only one transition for each different label. The *STG* can be obtained by petrify by using the command:

```
petrify vme.g -o vme.1label
```

The result is shown in Figure 6.3(b). However, this representation is much less readable that the previous one. For this reason we will use the *STG* of Figure 6.3(a) to synthesize the circuits presented in this chapter. With that representation, the specification of timing assumptions will become much more intuitive, since we will be able to independently define them for the READ and WRITE cycles.

## 6.2 Speed-independent implementation

Let us first derive a speed-independent implementation of the VME bus controller. The notation used for the representation of generalized C elements is shown in Figure 6.5.

After executing the command

```
petrify vme.g -gc -fr10 -eqn vme.eqn -log vme.log -o vme.csc
```

The option `-fr10` increases the width of the exploration when solving CSC conflicts. The solution obtained is depicted in Figure 6.6. One internal signal, *csc*0, has been inserted to encode the states.

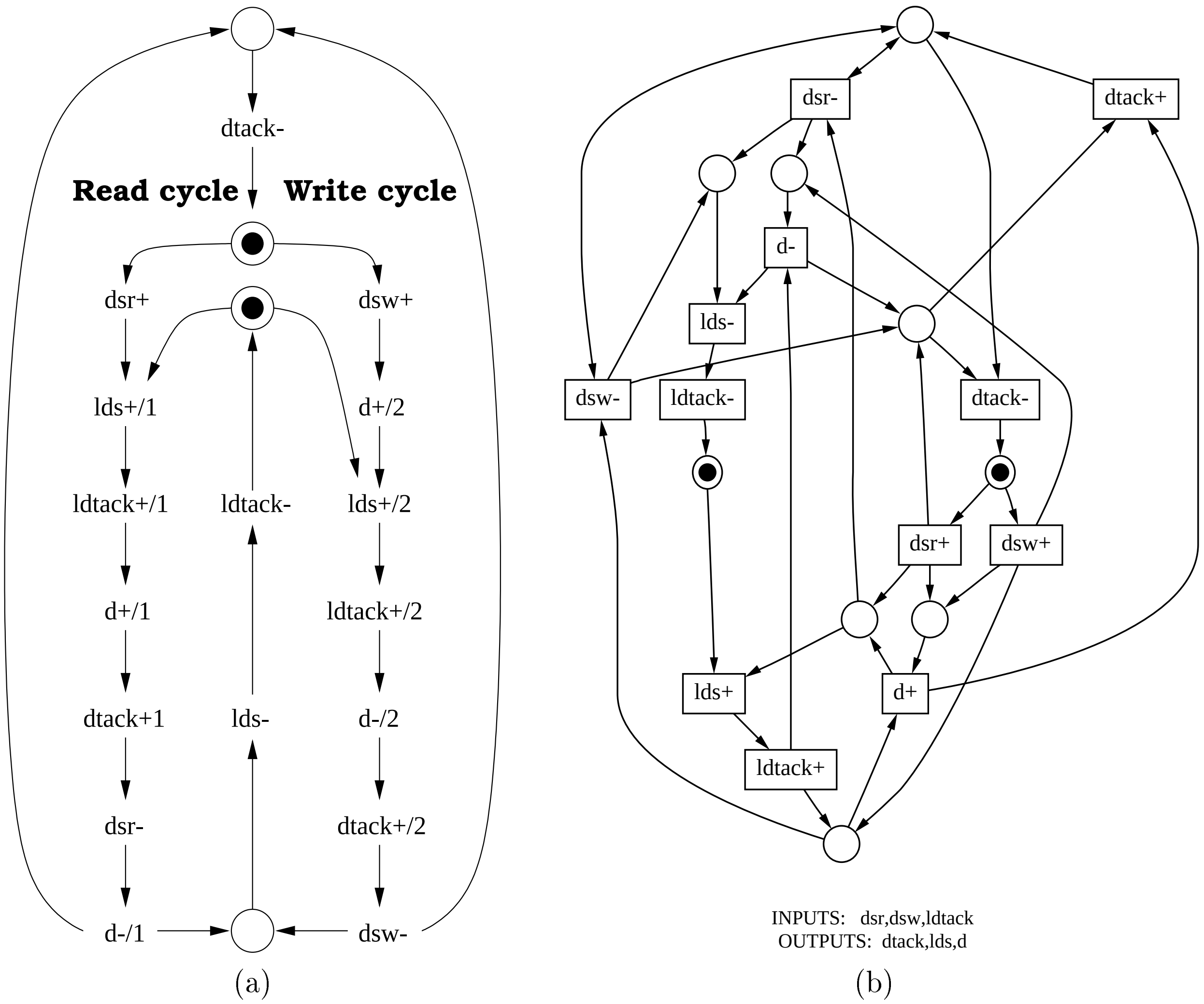



Figure 6.3: (a) *STG* for the VME bus controller. (b) Equivalent *STG* with only one transition for each signal event.

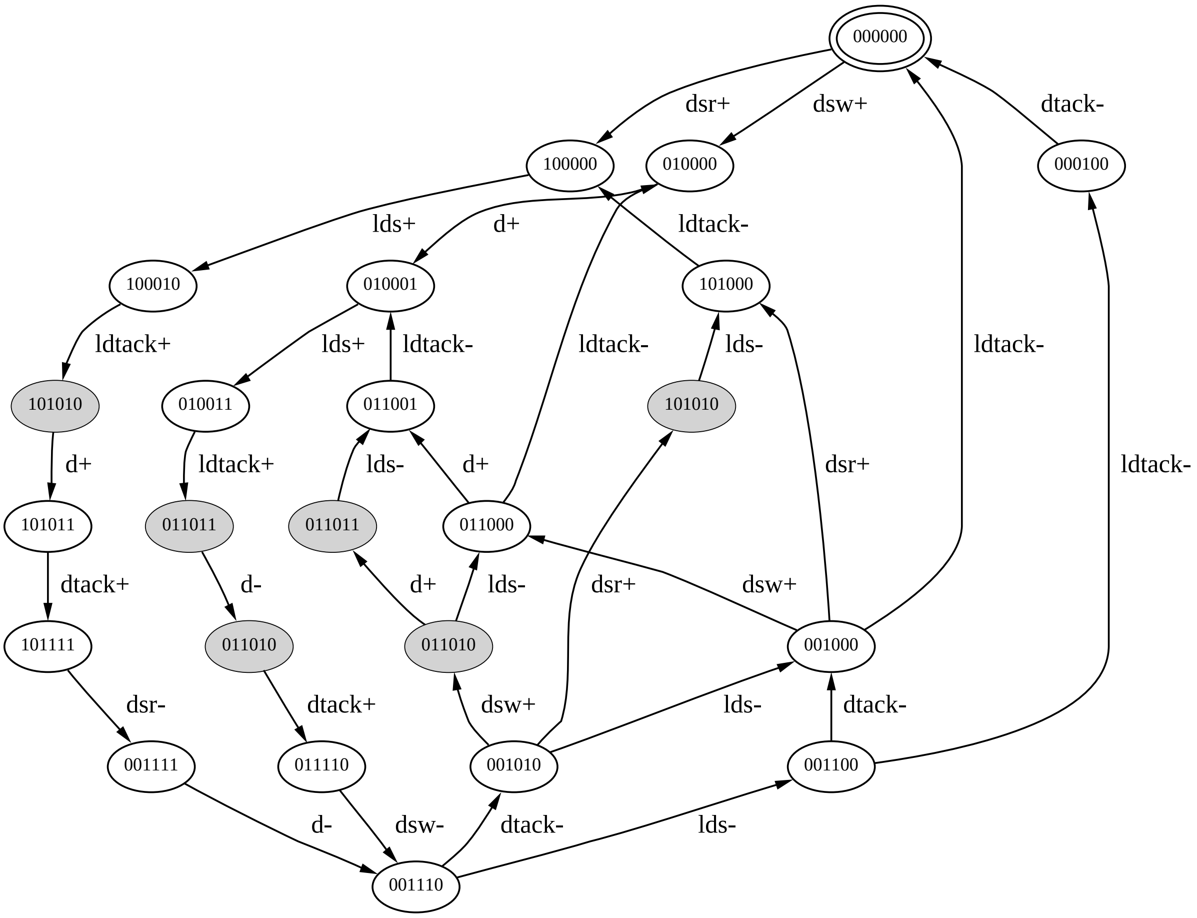



Figure 6.4: State graph of the VME bus controller.

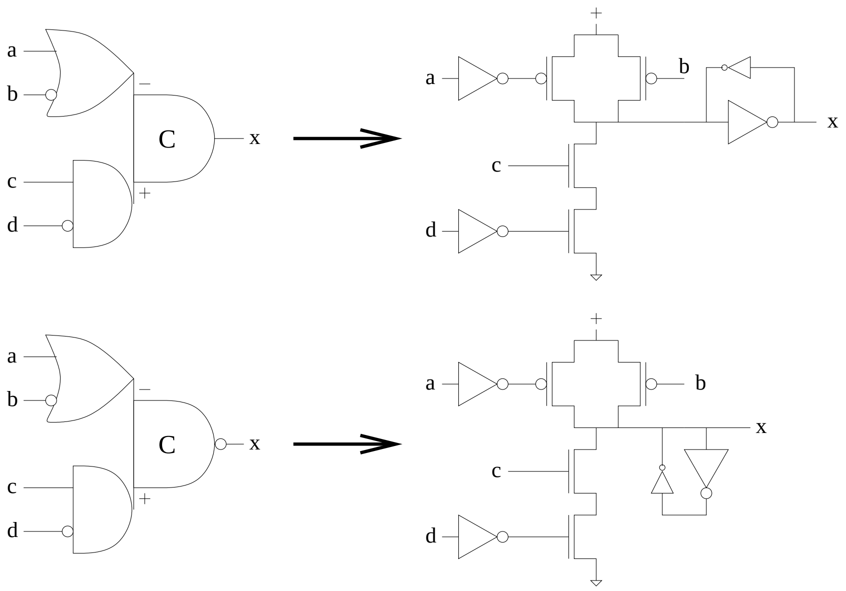



Figure 6.5: Notation used to represent generalized C elements.

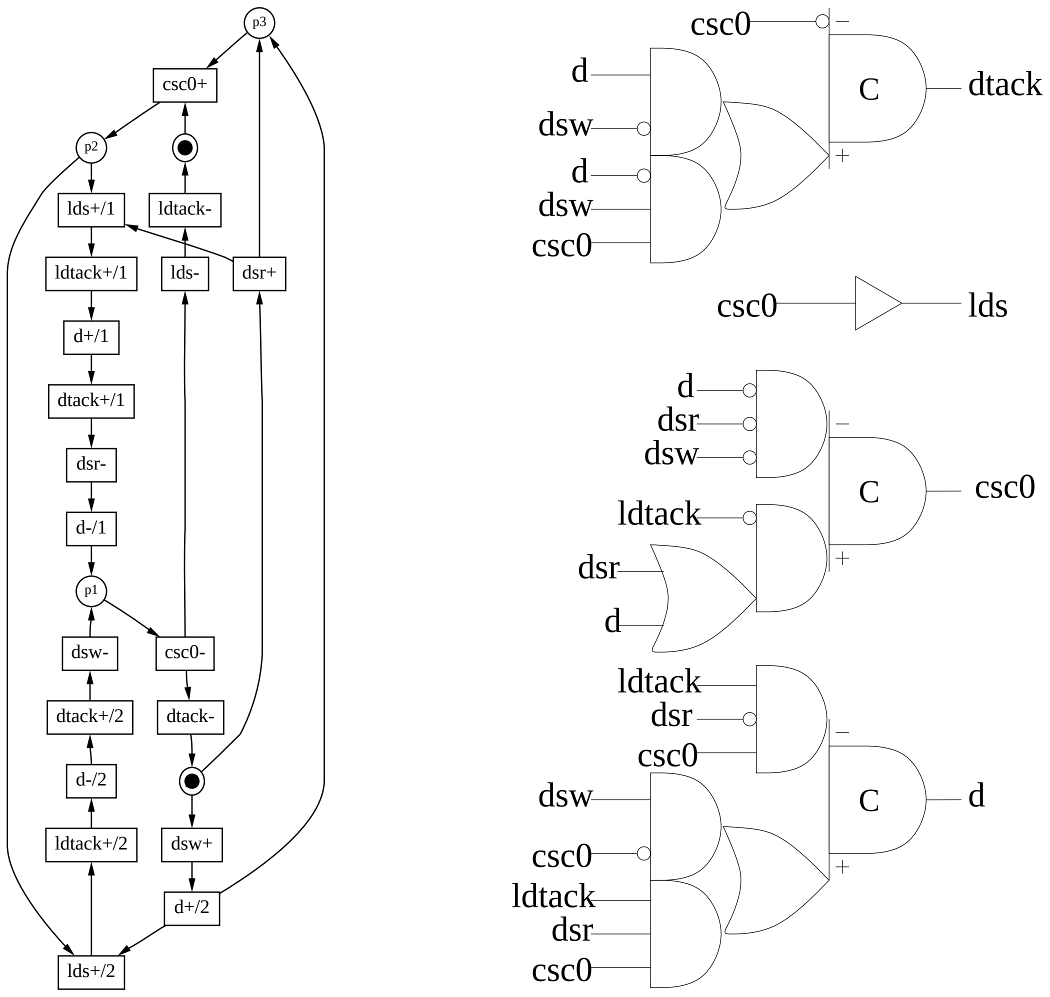



Figure 6.6: Specification (`vme.csc`) and speed-independent implementation of the VME bus controller.

```
.inputs dsr dsw ldtack
.outputs dtack lds d
.graph

p0 dsr+ dsw+
p1 lds+/1 lds+/2

# Read cycle
dsr+ lds+/1
lds+/1 ldtack+/1
ldtack+/1 d+/1
d+/1 dtack+/1
dtack+/1 dsr-
dsr- d-/1
d-/1 p2 p3

# Write cycle
dsw+ d+/2
d+/2 lds+/2
lds+/2 ldtack+/2
ldtack+/2 d-/2
d-/2 dtack+/2
dtack+/2 dsw-
dsw- p2 p3

# Return to zero
p2 lds-
p3 dtack-
lds- ldtack-
dtack- p0
ldtack- p1

.marking {p0 p1}
# Assuming an slow environment
.slowenv
.end
```

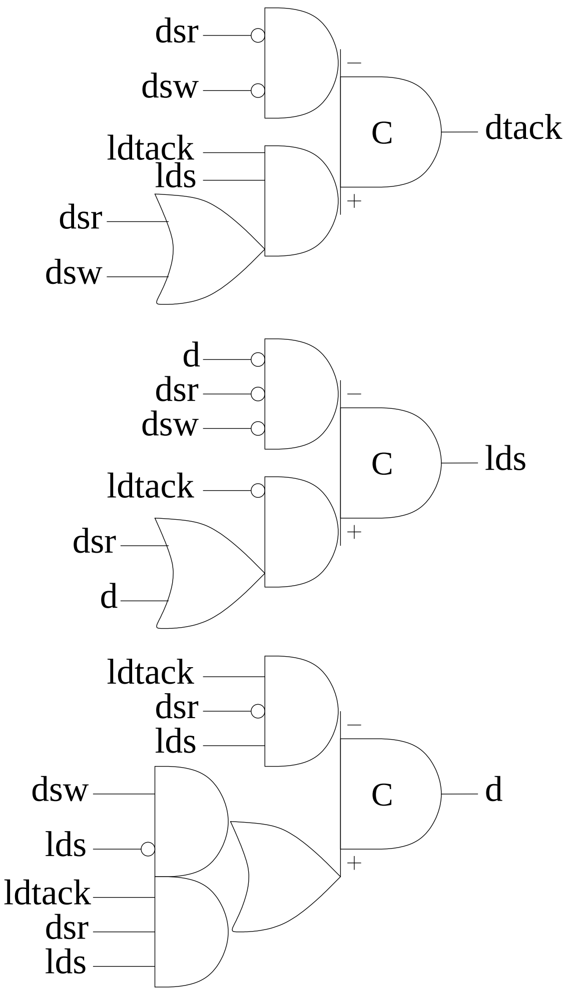



Figure 6.7: Specification and circuit under the assumption of a slow environment.

## 6.3 Assuming a slow environment

Let us now consider that the response time of the environment is long enough to enable the circuit complete its internal activity in progress. Such assumption can be exploitec by enabling petrify to generate automatic timing assumptions after having specified the input events as "slow". This can be achieved by including the `.slowenv` statement in the specification and using the option `-atopt` for automatic generation of timing assumptions:

```
petrify vme.g -gc -atopt -eqn vme.eqn -log vme.log -no
```

The result is shown in Figure 6.7.

Note that, in this solution, no state signals have been inserted to solve the state encoding problem. This is due to the fact that the timing assumptions make some states with conflicts unreachable. In this particular case, all conflicts dissapear.

In order not to make the explanation too tedious, we will skip now the timing analysis of the solution, but we will focus on the automatic timing assumptions generated by petrify. They are reported in the file `vme.log`:

```
.time lds-<|dsr+        # concurrency reduction (automatic)
.time lds-<|dsw+        # concurrency reduction (automatic)
.time lds-<|d+/2        # concurrency reduction (automatic)
.time dtack-<|lds-      # concurrency reduction (automatic & simultaneous)
.time dtack-<|ldtack-   # concurrency reduction (automatic)
.time lds-<|dsr+        # concurrency reduction (automatic)
.time lds-<|dsw+        # concurrency reduction (automatic)
.time dtack-<|ldtack-   # concurrency reduction (automatic)
.time lds+/2>d+/2       # early enabling (automatic)
.time dtack+/1>d+/1     # early enabling (automatic)
.time dtack+/2>d-/2     # early enabling (automatic)
.time lds->d-/1         # early enabling (automatic)
.time dtack->d-/1       # early enabling (automatic)
```

Note that the fourth assumption (`.time dtack-<|lds-`) corresponds to the ordering of two output events that are enabled simultaneous. The message `‘‘automatic & simulatenous’’` indicates two important things: (1) that the assumption has been generated automatically and (2) that both events are enabled simultaneously and petrify gave priority to one of them when deciding the firing order.

This is one of the cases where the intervention of the designer can be useful. What would be the solution if we would not assume any firing order or another firing order?.

The first possibility can be exploring by adding the following timing assumption in the specification: `.time lds-<>dtack-`, indicating no assumptions should be made on the firing order of both events.

The second possibility can be explored by adding the following timing assumption in the specification: `.time lds-<|dtack-`. Let us now execute the same command as above. The new reported timing assumptions in `vme.log` will be the following:

```
.time lds-<|dtack-      # concurrency reduction (specification)
.time lds-<|dsr+        # concurrency reduction (automatic)
.time lds-<|dsw+        # concurrency reduction (automatic)
.time lds-<|d+/2        # concurrency reduction (automatic)
.time dtack-<|ldtack-   # concurrency reduction (automatic)
.time lds-<|dsr+        # concurrency reduction (automatic)
.time lds-<|dsw+        # concurrency reduction (automatic)
.time dtack-<|ldtack-   # concurrency reduction (automatic)
.time lds+/2>d+/2       # early enabling (automatic)
.time dtack+/1>d+/1     # early enabling (automatic)
.time dtack+/2>d-/2     # early enabling (automatic)
.time lds->d-/1         # early enabling (automatic)
.time dtack->d-/1       # early enabling (automatic)
```

Note that the first timing assumption (`.time lds-<|dtack-`) comes now from the `‘‘specification’’` and precludes petrify to generate the assumption `.time dtack-<|lds-` that would contradict

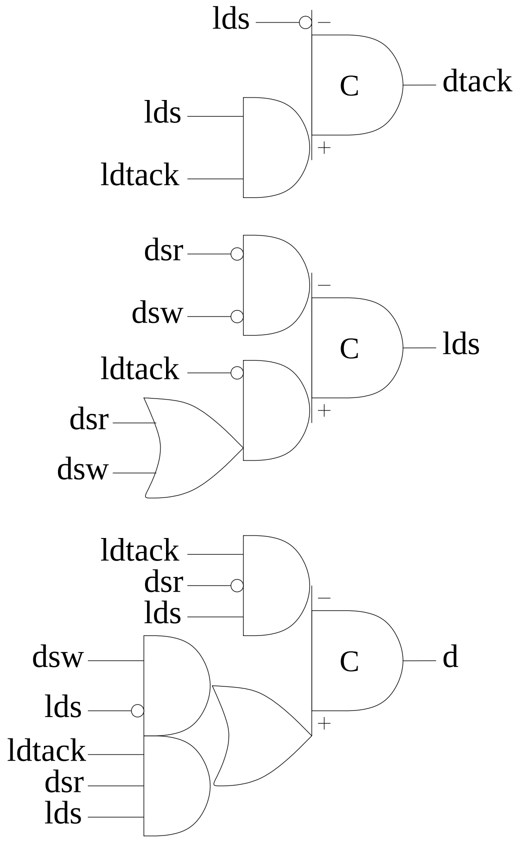



Figure 6.8: Circuit under the assumption of slow environment and `lds- <| dtack-`.

the first one. The circuit is shown in Figure 6.8 and confirms the possibility of improvement by reversing the firing order of `dtack-` and `lds-`. Again we will skip the timing analysis of this new solution.

## 6.4 Assuming a slow bus control logic

Looking at the specification of the controller in Figure 6.3(a) we can observe that the return-to-zero of the protocols at both sides of the controller (bus and device) is done concurrently.

Let us consider now, that we know more details about the speed of the control logic at the side of the bus and that we can deduce that it is slow enough such that any new request for a read or write cycle ($dsr+$ or $dsw+$) will never arrive at the controller before the handshake with the device has been completed. This can be specified by adding two new timing assumptions:

```
.time ldtack- <| dsr+ ldtack- <| dsw+
```

The resulting circuit is shown in Figure 6.9.

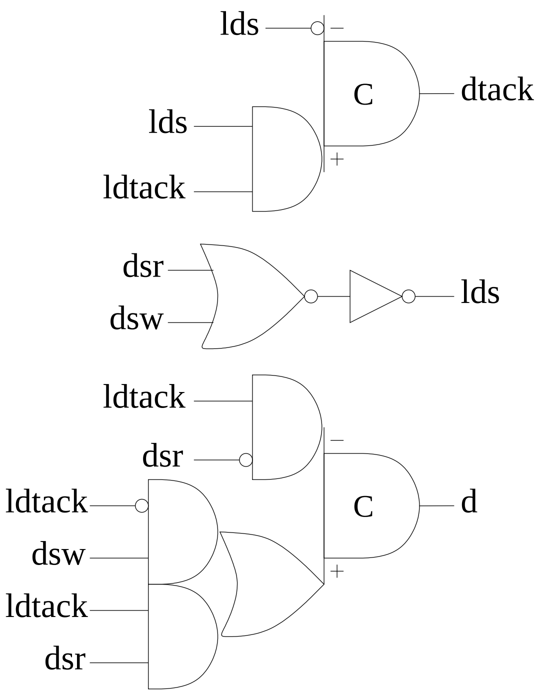



Figure 6.9: Circuit under the assumption of a slow bus control logic.

## 6.5 Timing analysis

Let us now make the timing analysis of the final solution depicted in Figure 6.9. For this, we will first study the information reported in the file `vme.log` about each gate.

```
SET(dtack')   = lds'
RESET(dtack') = ldtack lds
[dtack] = dtack'        (output inverter)
  > triggers(SET):     lds- -> dtack-
  > triggers(RESET):   ldtack+/1 -> dtack+/1   ldtack+/2 -> dtack+/2
  > 5 transistors (3 n, 2 p)
  > Estimated delay: rising = 23.96, falling = 16.00
  > Concurrency reduction:  lds-==>dtack-
  > Timing assumptions (early enabling):  d+/1<dtack+/1  d-/2<dtack+/2

...

lds' = dsw' dsr'
[lds] = lds'        (output inverter)
  > triggers(SET):     (dsr-,dsw-) -> lds-
  > triggers(RESET):   dsr+ -> lds+/1   dsw+ -> lds+/2
  > 6 transistors (3 n, 3 p)
  > Estimated delay: rising = 19.58, falling = 24.96
  > Timing assumptions (concurrency):  ldtack-<dsw+  ldtack-<dsr+
  > Timing assumptions (early enabling):  d+/2<lds+/2  d-/1<lds-
```

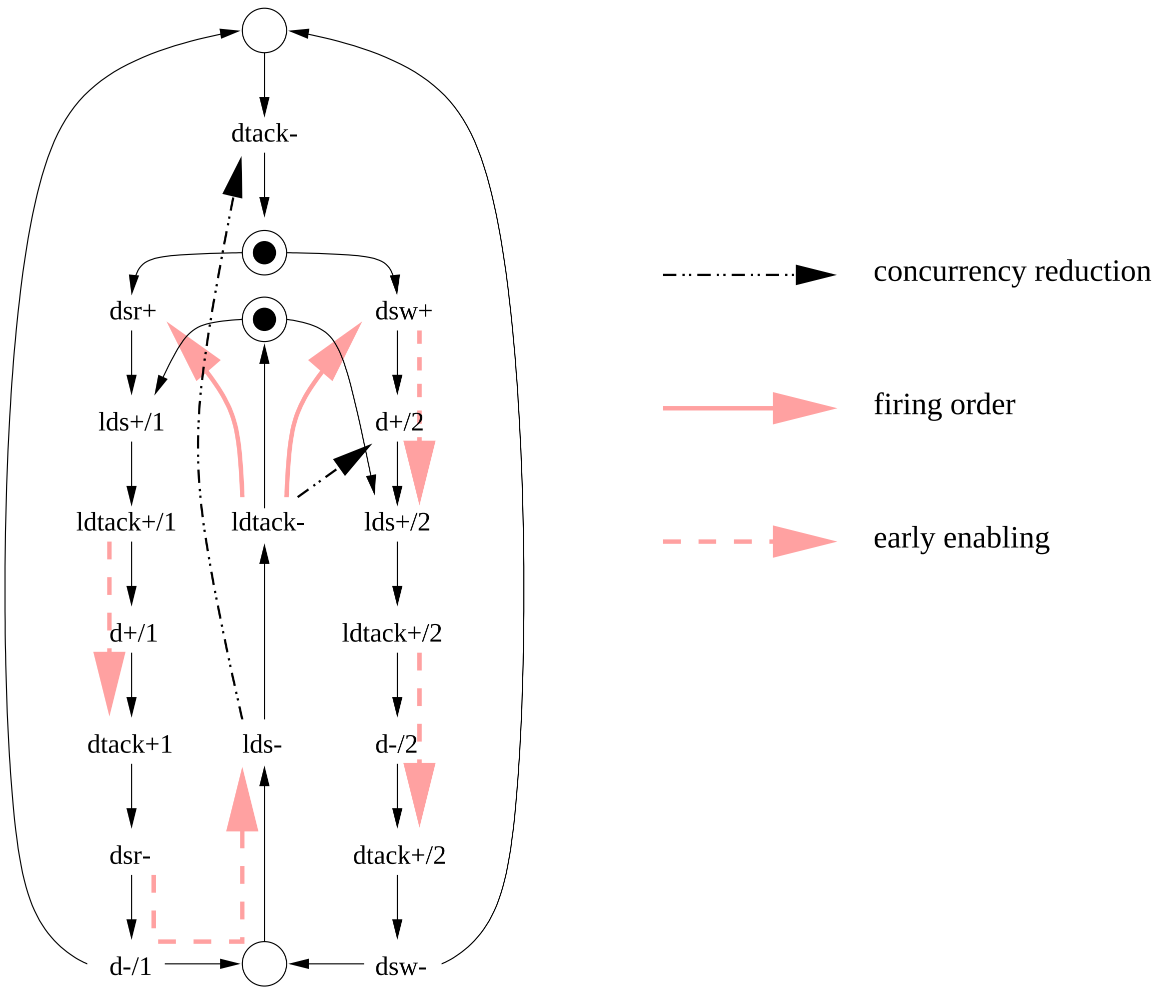



Figure 6.10: Timing assumptions for the final solution of the VME bus.

```
...

SET(d')   = dsr' ldtack
RESET(d') = dsw ldtack' + dsr ldtack
[d] = d'        (output inverter)
  > triggers(SET):     dsr- -> d-/1   ldtack+/2 -> d-/2
  > triggers(RESET):   ldtack+/1 -> d+/1   (dsw+,ldtack-) -> d+/2
  > 10 transistors (6 n, 4 p)
  > Estimated delay: rising = 31.33, falling = 29.08
  > Concurrency reduction:  ldtack-==>d+/2
  > Timing assumptions (concurrency):  ldtack-<dsr+
```

The timing assumptions are represented and summarized in Figure 6.10. We can distinguish three types of timing arcs in that diagram:

**Concurrency reduction arcs** that denote the additional causality relations enforced by

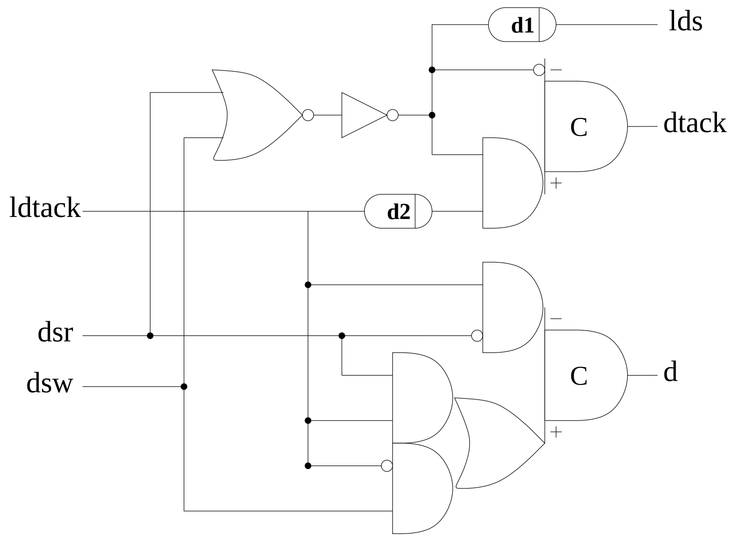



Figure 6.11: Delay padding to satisfy timing assumptions.

the logic ($lds- \rightarrow dtack-$ and $ldtack- \rightarrow d+/2$). These are not timing assumptions that must be verified for the circuit to be correct.

**Firing order arcs** that denote the assumed firing ordering of concurrent events for the circuit to be correct. These are timing assumptions that must be verified or enforced by delay padding.

**Early enabling arcs** that indicate the new trigger events for the early enabled events. For example, event $lds-$ that is triggered by $d-/1$ in the read cycle is now triggered by $dsr-$. This information is exrtacted by combining the early enabling timing assumptions and the list of trigger events reported for each solution. These are assumptions that must be verified for the circuit to be correct. In the mentioned example, it must be verified that event $d-/1$ will occur before event $lds-$.

$ldtack- \rightarrow dsr+$ and $ldtack- \rightarrow dsw+$ were the assumptions that we made about the speed of the bus control logic. Therefore, they can be considered as satisfied. However, the assumptions on early enabling of events must be carefully analyzed, since their validity depends on the actual delays of the derived logic.

All the early enabling assumptions rely on the fact that the delay of gate $d$ is shorter than the delay of $lds$ and $dtack$. Unfortunately, we see that gate $d$ is more complex than the other gates, so these assumptions can be considered as unrealistic. Therefore, some changes must be performed on the circuit to satisfy those assumptions.

A possible solution is depicted in Figure 6.11. The delay $d_1$ should ensure that $lds+/2$ and $lds-$ will fire later than $d+/2$ and $d-/1$ respectively, even they are triggered simultaneously. The delay $d_2$ should ensure that $dtack+/1$ and $dtack+/2$ will fire later than $d+/1$ and $d-/2$ respectively. Note that the chosen solution only delays the triggering effect of signal $ldtack+$ on $dtack+$. Another solution would have been to add a delay at the output of gate $dtack$, but this would also delay $dtack-$, which is not necessary for the correct functioning of the circuit. Thus, we can see that the information reported by petrify allows to customize the circuit such that timing assumptions are met for individual events of a signal, rather than for all the events of the signal.