Criteria of Endo-/Isochrony in Quartz

Bachelorarbeit von

Daniel Thielsch

10. Mai 2012

Referent: Prof. Dr. Klaus Schneider

Betreuer: Jens Brandt

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Criteria for Endo-/Isochrony in Quartz" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 11. Mai 2012

(Unterschrift)

Daniel Thielsch

Abstract

This thesis concentrates on the criteria for classifying endo- and isochronous systems while defining what is meant by running any synchronous Quartz program in an asynchronous environment. For that reason a series of abstractions as well as the principles of endo- and isochrony are defined. By the help of these definitions a set of criteria is in detail proposed and proofed.

Zusammenfassung

Diese Arbeit konzentriert sich auf die Kriterien zur Klassifizierung endo- und isochroner Systeme unter Berücksichtiung der Fragestellung, was damit gemeint ist ein synchrones Quartz program in einer asynchronen Umgebung auszuführen. Aus diesem Grund wird eine Reihe von Abstraktionen wie auch die Prinzipien der Endo- und Isochronie definiert. Mit Hilfe dieser Definitionen werde eine Reihe von Kriterien im Detail eingeführt und bewiesen.

Contents

1 Introduction **9**
1.1 Motivation . . . 9
1.2 Outline . . . 10

2 Related Work **11**
2.1 References . . . 11
2.2 The Synchronous Paradigm . . . 11
2.3 Quartz . . . 13
2.3.1 Syntax and Informal Semantics . . . 13
2.3.2 Semantical Challenges . . . 15
2.4 Synchronous Guarded Actions . . . 16

3 Endochrony and Isochrony **19**
3.1 From Synchrony to Asynchrony - GALS . . . 19
3.2 Asynchronous Guarded Actions . . . 20
3.3 Basic Definitions . . . 25
3.4 Endochrony . . . 30
3.4.1 Definition . . . 30
3.4.2 Latency Insensitivity . . . 34
3.4.3 Criteria . . . 35
3.5 Isochrony . . . 43
3.5.1 Definition . . . 43
3.5.2 Criteria . . . 45

4 Summary **52**
4.1 Conclusion . . . 52

"Es braucht viel Zeit, einen kurzen Weg zu gehen"

Antigone

1 Introduction

1.1 Motivation

After the synchronous paradigm had been proposed back in the early 80's, it did not take long for the first synchronous languages to arrive. Building on the foundations of the synchronous paradigm, by combining both synchrony and concurrency, languages like Lustre or Esterel[3, 4] were frequently used to model safety critical embedded systems or digital circuits. Although specification and verification is simplified, the growing popularity for distributed systems has led to certain problems. Particularly varying timing or computational delays of a system cause problems, since the synchronous paradigm entails a global notion of time. To overcome these problems, the globally asynchronous locally synchronous (GALS)[17, 13] architectures were developed. As the name suggests, GALS systems comprise locally synchronous components, connected via asynchronous communication lanes. This allows each component to run at its own speed thereby retaining the correctness of all computations. Nevertheless, GALS architectures do not come free of charge. Not every synchronous component can be deployed in an asynchronous environment. Only components being able to account for the asynchrony, by resynchronizing all incoming input values are suited for the needs of a GALS architecture. Moreover all deployed components must ensure that despite the asynchronous environment every computations is executed as a synchronous architecture would do. The corresponding concepts are called endochrony[1] respectively isochrony[16]. For the programmer a problem arises if he wants to tell whether one of his synchronous components meets one of the requirements or not[14, 15]. Being defined in terms of behaviours, both concepts do not provide simple syntactic criteria to tell if either concept is fulfilled. The question remains, if no set of criteria can be found , sufficient to tell if a component written in a specific synchronous language does satisfy the concepts of

endo-/isochrony. This is the question , I will address in the following.5

1.2 Outline

Before starting, I give a brief outline of the topics to come. In Chapter 2, I will shortly sketch the synchronous paradigm and its applicability to asynchronous environments. Based on a certain type of asynchronous environment, Chapter 3 will introduce the concepts of endo-/isochrony as well as the criteria necessary for classifying any synchronous system. Finally Chapter 4 concludes the thesis by providing future work.

2 Related Work

2.1 References

As most of this thesis builds upon the foundation laid in [1], it provides a good starting point. The most comprehensive elaboration of these topics is likely to be found in [18], including a wealth of examples and donating some of its definitions to this thesis. GALS architectures are largely discussed in [13, 6], whereas [9, 10] defines a form of endochrony, slightly different from the one introduced in [1]. In [11], the work of [1] has been refined, now with a distinct focus on concurrency in synchronous processes. Though not directly mentioned in this work, a set of asynchronous guarded actions is likely to be concurrent, as is the process derived from this set. For a broad overview [2] is good commendation, since it reviews the most popular synchronous languages and provides insights in current issues and problems.

2.2 The Synchronous Paradigm

Considering classes of computer systems, three essential system types can be distinguished: transformational systems and interactive/reactive systems. While transformational systems perform a transformation, from a set of inputs to a set of outputs, once, interactive/reactive systems constantly interact with the environment. Though both share continuous interaction with the environmental system, interactive systems initiate interaction on their own. Quite the contrary holds for reactive systems, which listen for the environment to trigger its execution. If necessary it is possible to abstract from those environmental triggers, by assigning each interaction a logical point of time. Those logical points of time do not only allow to abstract from any physical time but they also provide an ordering of interaction events. Each interaction event is additionally considered to be instantaneous. This means, that it does not take

time for reactive system to serve any incoming interaction request. Thinking of each computation as some reaction, everything necessary to grasp the essential idea of the synchronous paradigm has been given. The synchronous paradigm describes systems [11]:

1. Evolving through an infinite sequence of successive reactions indexed by a global clock
2. Computing for each component during a reaction its output signals based on its internal state and input values
3. Propagating all events between components *synchronously* within the same reaction

Looking at the preceding description it should be clear why the synchronous paradigm is often used for modelling reactive systems. Both do share the characteristic of dividing the overall system behaviour into a sequence of reactions. Each reaction is thereby carried out by using environmental inputs and internal system states to compute some output. However this output does not have to be computed by a single component alone. During a reaction a multitude of components can interact with each other, reading and submitting values, until the system outputs are computed at last. Even though components produce output values in each reaction, the synchronous paradigm does not force them to assign each output variable a value. While this might pose a problem at first glance, it easily handled by remembering the assumption that components interact synchronously. The absence of a variable value is therefore bound to a particular reaction which allows for some way of handling the absence. To model such behaviours, the domain of every variable is extended by the special value $\perp$, named absence. As the absence of a variables value can be detected and used for controlling ongoing computations one might think of $\perp$ as an actual value, read by all receiving components for further progress. On the top of all this resides a global clock, helping to keep track of the system by monitoring the infinite sequence of progressing reactions. The combination of concurrent components acting synchronously at each *clock tick* is often referred to as "deterministic concurrency". Timing issues do not play any role, such that the programmer can concentrate on specifying and analysing the semantics of his system. Although the paradigm does abstract from most low level

details, they do come into play when implementing the synchronous program on its target architecture. As a matter of fact all interacting components of a system must have finished executing when the next clock tick of the global clock occurs. If that is not the case, the system runs into trouble. Therefore, the global clock must account for any timing delays between connected components or different computation times, e.g. for running at the speed of the slowest component.

2.3 Quartz

2.3.1 Syntax and Informal Semantics

Following the synchronous paradigm means writing programs in a synchronous language, that inherits the synchronous approach. Although there are many synchronous languages available[7], I will consider only one particular imperative synchronous language, named Quartz. Quartz is in many ways similar to the synchronous language Esterel and can be characterized as a derivative. It comes with a host of features like delayed assignments and asynchronous parallel execution of threads that clearly surpasses Esterel's expressiveness in modelling synchronous behaviours. The main features of Quartz can be easily grasped knowing the basics of Esterel. The same holds for the program execution. Every program execution consists of a series of cycles, called macro-steps. Executing a macro-step means reading all inputs, executing a set of assignments, called micro-steps, simultaneously and returning all the outputs. While micro-steps do not take time, macro-steps do. This is indicated in the program code by the statement "pause". In the case of a program execution the control flow moves therefore from pause to pause statements. While the program execution comes to a halt reaching a *pause* statement, *nothing* does not change anything, neither data- nor controlflow are influenced by this statement.

The rest of the core statements, given in Table 2.1, can be explained as follows: For assigning variable values either an instantaneous assignment $x = \tau$ or a delayed assignment $next(x) = \tau$ can be used. The difference is, while the instantaneous assignment is carried out in the same macro-step, the delayed assignment is evaluated in the current macro-step, but executed in the

statement	explanation
nothing	empty statement
l:pause	new macro step
x = τ, next(x) = τ	initial/conditional assignment
if(σ) S_1 else S_2	conditional
$S_1; S_2$	sequence
do S while(σ)	loop
[weak] [immediate] abort S when(σ)	abortion statements
[weak] [immediate] suspend S when(σ)	suspension statements
$S_1 \vert\vert S_2$	synchronous concurrency
{α x; S}	local variable declaration

Table 2.1: Core Quartz statements

next one. As for the conditional statement, it does what its name suggests and checks for a given boolean expression σ if the if-clause σ is evaluated to true. If so S_1 is executed, otherwise S_2. Multiple statements are concatenated by the use of the semicolon ; and repeated execution of statements S is done by a *while loop*, as long as σ holds. However it is required, that S contains at least one reachable pause statement, such that even the macro-step of an infinite loop will come to a halt. The most unusual statements are the *abort* and *suspension* statement. Without any *weak* or *immediate* modifier, the *abort* statement checks, when entered, in each new macro-step if σ holds. If so, it does not execute any statements left of S and proceeds with statements following the abort statement. Otherwise it simply executes statements S, until finished. Very much alike the abort statement is the *suspend* statement. While it also checks for each new macro-step if its clause σ holds, it does not abort the execution of statements but cause the control-flow to stay at its current location in S. In addition, *weak* and *immediate* modifiers allow a finer execution control of those statement, by specifying which statements to execute if σ holds or when σ is checked. The exact details of these modifiers will be omitted at this point as they are of no importance for this thesis. Finally, operator $\|$ denotes that for a statement $S_1 \| S_2$, both statements are run by separate threads in lockstep, while $\{\alpha\ x;\}$ denotes a local variable declaration. Notice, as many threads can be active during a macro-step, not

only one but several pause statements can be reached after the macro-step has finished executing. Besides its primary purpose to indicate the consumption of logical time, pause statements therefore serve as a point of synchronization, used by threads to synchronize their execution.

2.3.2 Semantical Challenges

In the section 2.2 it was mentioned that even though each reaction does read inputs and compute outputs, not necessarily all in-/outputs must carry a value. How does this fit with the behaviour of Quartz to read all inputs for computing all outputs? As it was stated before, the current set of variable values is bound to a specific reaction. This allows the component to know if some value is available during a reaction or not. But since "absence" cannot be used for computation, some value must be used and this is determined by the "reaction of absence". Depending on the data type of the variable, two alternatives are possible. For event variables the default value is used and for memorized variables, the value of the last reaction is used. Furthermore, it is necessary to ensure that each variable carries a unique value within a macro step. Due to perfect synchrony, a block of statements like

$$pause;\ x = 1;\ x = 2;\ pause;$$

is not allowed. It is a different thing with the following block of statements

$$pause;\ x = 1;\ next(x) = 2;\ pause;$$

Although it has two assignments, the assignment for x = 2 is not executed in the current macro step but in the next one. All statements evaluated in the current macro step, but executed in the macro step ahead are called delayed assignments. They are perfect example for the way Quartz builds on the foundation of Esterel by enhancing its capabilities in an intuitive manner. What's left is for the programmer to check if his program is causally correct[3]. A program is considered causally correct if *'it assures that for all reachable states and all possible inputs, the micro steps can be executed in an ordering where all values are known when needed'* [18]. It seems appropriate to clarify this definition by using a small example. The program

```
module T01(event o) {
 if(o)
    emit(o);
}
```

is satisfied by both $o = true$ and $o = false$. For the case $o = true$, execution jumps right into the if clause and executes the assignment o=*true*[1].On the other hand for o=*false*, the if-clause is skipped and the program exits. Before this happens the execution scheme acknowledges that$o$ has not received any value. In consequence "reaction to absence" is triggered, assigning o the boolean default value FALSE. Although the program does have a behaviour, it is not constructive, since non-determinism during runtime is not allowed. The same holds for the program

```
module T02(event o) {
 if(!o)
  emit(o);
}
```

In this case the program does not even have a behaviour, rendering it unconstructive as well. Those two examples are representative of a whole class of synchronous programs that fail to meet the requirements for causal correctness and are therefore disregarded.

2.4 Synchronous Guarded Actions

By now it should be evident from the way statements are executed that control- and dataflow can be modelled using a abstract finite state machine. Interpreting the combination of pause statements as states, the execution of a macro step reads inputs, computes outputs and leads finally again into a state. While a finite state machine might be pretty handy, it often suffers from large blow-ups, considering the fact that a synchronous program can run multiple threads and be at several pause statements at once. Since this greatly extends the size of the state space, an alternative had to be found. This was done by encoding the control- and dataflow in terms of synchronous guarded actions.

[1] emit(x) is a shortcut for x=true

Encoding the control- and dataflow is not as easy as it might look like at first. As can be seen by looking at a synchronous program, the execution of a macro-step depends on previous states and input values. Although these values might change each cycle, the conditions that impose the variable values do not change. This allows for each variable, including the pause labels, to derive a boolean expression that determines when the action, the assignment of a certain variable value, is carried out. A conditional variable assignment is called guarded action. For a set of variables V, a synchronous guarded action γ_{sy} has one of the following forms:

- $\gamma_{sy} := g(y_1, \ldots, y_n) \Rightarrow x := f(z_1, \ldots, z_m)$
- $\gamma_{sy} := g(y_1, \ldots, y_n) \Rightarrow next(x) := f(z_1, \ldots, z_m)$

where $g : \mathbb{B}^n \rightarrow \mathbb{B}, f : \mathbb{B}^m \rightarrow \mathbb{B} \wedge \forall_{i,j \in \mathbb{N}}\ y_i, z_j, x \in V$.

While the left hand side, represented by the boolean function g is called guard, the right side, the assignment $x/next(x) := f(\vec{z})$, is called the action. Action and guard form the so called guarded action, that is to say, if a guard has been evaluated to true, the action is instantaneously executed. Having a set of those guarded actions, if a clock tick occurs, all synchronous guarded actions are executed synchronously. In the course of this work it will be often the case that certain constraints apply only to parts of a guarded action, such that the following definitions are needed:
For given $P' \subseteq S := \{\gamma \mid \gamma \text{ is a guarded action}\}$ and $\gamma \in S$:

$Var(f) := \{x | x \in V \wedge \text{x is a identifier in f}\} \wedge f : (\mathbb{B} \cup \{\bot, \top\})^n \rightarrow (\mathbb{B} \cup \{\bot, \top\});$

$grd(\gamma = [g(\vec{y}) \Rightarrow x := f(\vec{z})]) := g(\vec{y})$
$act(\gamma = [g(\vec{y}) \Rightarrow x := f(\vec{z})]) := f(\vec{z})$

$grdVar(\gamma) := Var(grd(\gamma))$
$rdVar(\gamma) := Var(act(\gamma))$
$wrVar(\gamma = [g(\vec{y}) \Rightarrow x := f(\vec{z})]) := \{x\}$
$nwrVar(\gamma = [g(\vec{y}) \Rightarrow next(x) := f(\vec{z})]) := \{x\}$

$$grdPa(\gamma) := \{\alpha_i | grd(\gamma) := \bigvee_{i:=1}^{n} \alpha_i; n \in \mathbb{N} \land \alpha_i : \mathbb{B}^{|\alpha_i|} \to \mathbb{B}\}$$
$$sgrdPa(\gamma) := \{\alpha_i | \alpha_i \in grdPa(\gamma) \land \exists r \in Rct(P).[\alpha_i]_r = true\}$$
$$P_x := \{\gamma | \gamma \in P' \land wrVar(\gamma) = \{x\}\}; x \in wrVar(P')$$

The sets $grdVar$, $rdVar$ and $wrVar$ return the set of variables that occur in the respective part, while functions *grd* and *act* return the guard and action of a set of guarded actions. It is different to $grdPa$ and $sgrdPa$. As every guarded actions is assumed to be in DNF, $grdPa$ provides the set of clauses linked by the $\land$-operator and $sgrdPa$ provides those clauses that are actually satisfiable. Moreover all definitions are naturally lifted to sets of guarded actions. Given now any constructive Quartz program, the entire program can be encoded by a set of synchronous guarded action[2]. Executed by a synchronous interpreter, that handles reaction to absence, they exhibit the very same behaviour as the original Quartz program. While it is clear how to run a set of synchronous guarded actions in a synchronous environment, the question remains how to run them in an asynchronous environment. Therefore it is time to look at asynchronous architectures and the resulting consequences.

[2]For further details on the encoding, see [18]

3 Endochrony and Isochrony

3.1 From Synchrony to Asynchrony - GALS

So far only synchronous architectures were considered. It goes without saying, that synchronous architectures come with certain disadvantages. To get rid of those disadvantages, a different class of architectures, the distributed ones, will be considered next. Distributed architectures are composed of multiple components, connected via a network. The eventual usage of buffers allows each component to run by its own clock and spares them the necessity to synchronize its behaviour with neighbouring components. Trying to define a global clock for the overall system will therefore fail, making it hard if not impossible to run a synchronous program on such a system without any additional effort. Yet there is a way to get some of them running at least. Taking a look at the synchronous architectures shows that many of them are also composed of different entities. If there was a way to assign each entity to an synchronous component and ensure that this component is only triggered if all necessary inputs are available, a very efficient implementation would be the result. Since every component can run as soon as all inputs are available it is no longer restricted to wait for the next upcoming clock tick. An architecture that runs as suggested is called a GALS architecture[6]. Locally clocked synchronous components connected by asynchronous communication channels. However there is a problem associated with ensuring that each component is only triggered with its right inputs[1]. The fact that GALS's employ asynchronous communication makes it impossible to establish a global clock for the system, rendering the notion of a reaction obsolete. However without any reaction the absence of variables values does not have a meaning either. Only the sequence of values send and their ordering matters. Key for resolving this issue is to consider

[1] "Right" in the sense of how a synchronous architecture would trigger the component.

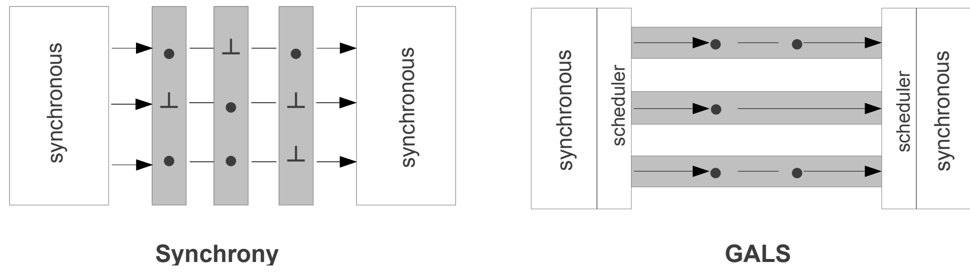

Figure 3.1: Information flow in synchronous and asynchronous environments (dots representing boolean values)

synchronous programs, that can account for this loss of synchronization information. Hence every component must have a wrapper that resynchronizes all incoming values and constructs synchronous inputs, if such a wrapper does not exist the synchronous program will not have a run. The only way then to handle this cases is it to add complementary synchronization values that show if some variables are present or not. Unfortunately, this solution results in increased communication and slows down computation speeds, contrasting ongoing efforts for efficiency. As a consequence the primary focus lays on synchronous programs having wrappers and the criteria for the existence of those.

3.2 Asynchronous Guarded Actions

Recalling the initial motivation, it seems clear that synchronous guarded actions are not well suited for describing asynchronous behaviours, since they heavily rely on the reaction of absence. Therefore it seems to be a good idea to discard the idea of an reaction to absence and to define a asynchronous guarded action γ_{asy}, that works with the value $\perp$ for evaluation and has the form:

- $g(y_1, \ldots, y_n) \Rightarrow x := f(z_1, \ldots, z_m)$

where $g, f : (\mathbb{B} \cup \{\bot, \top\})^{n|m} \rightarrow (\mathbb{B} \cup \{\bot, \top\}); \forall_{i,j \in \mathbb{N}}\ y_i, z_j, x \in V$

The set of all feasible guarded actions feasible is thereby denoted by S. This time the asynchronous guarded action is not allowed to write variables at the next clock tick. As asynchronous environments abstract from reactions, writing to a "next" clock cycle is meaningless. Nevertheless, one could argue that it should be allowed to use next assignments, if a write variable can *only* be written by next assignments. For the sake of simplicity such cases will be forbidden and only instantaneous assignments will be used. Furthermore it will be assumed that each guard and action of a guarded action is in disjunctive normal form (DNF). As there is always an equivalent DNF for a boolean function, this is no real constraint and helps in proving criteria further on. While a synchronous guarded action is only defined over boolean values, a asynchronous guarded action can also deal with values $\bot$ and $\top$. Where $\bot$ indicates that a variable is absent, the value $\top$ shows that a variable value is invalid. Most of the times this is caused a by a write conflict. Those new values are used in the context of boolean operators, hence it is necessary to define truth tables 3.2 for operators $\neg, \wedge$ and $\vee$ to define how any asynchronous guarded action is evaluated. No additional truth tables are needed, since the boolean functions of any guarded action must be in DNF. Though asynchronous guards do not

$\vee$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	1	$\top$
0	$\bot$	0	1	$\top$
1	1	1	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\wedge$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	$\bot$	$\top$
0	0	0	0	$\top$
1	$\bot$	0	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

x	$\neg x$
$\bot$	$\bot$
0	1
1	0
$\top$	$\top$

Figure 3.2: Truth tables for values $\cup\{\bot, \top\}$ and operator basis $\{\neg, \wedge, \vee\}$

rely on a reaction to absence, they are still embedded in a synchronous environment, that is clock driven. For any such clock tick, the set of guarded actions can be regarded as a reaction, a function $r : V \rightarrow \mathbb{B} \cup \{*, \bot, \top\}$ that assigns all variables a value of $\mathbb{B} \cup \{*, \bot, \top\}$. Because any reaction is defined

over all variables, the value $*$ is used to indicate that a variable is not used, being the case if it does not occur in any guarded action. The signature of a reaction r is then defined as $sig(r) := \{v \mid r(v) \notin \{*, \top\}$, the set of variables actually valid for a reaction. Since a set of guarded actions P employs only a certain set of variables, it holds that $r \in Rct(P) \Leftrightarrow: sig(r) = Var(P)$, yielding with Rct(P) the set of potential reactions. In addition, for any reaction r and boolean function f, $[f]_r$ denotes the evaluation of f, assigning the values of r. Given now any set of guarded actions P, the possible behaviours of P are defined in terms of all possible reactions, namely

$$Bhv(P) := \{(r^t)_{t \leq n} \mid n \leq \infty \wedge \forall_{t \leq n} i.sig(r^t) = Var(P)\}$$

By definition is seem clear that a behaviour is nothing more than a sequence of reactions. Hence it will be often useful to consider such a sequence of reactions $r_1 \cdot r_2 \cdot \epsilon$, where r_1, r_2 are reactions at time some time points t, t+1 and ϵ is a behaviour of undefined length. Furthermore, as from that point on, only asynchronous guarded actions will be considered, as the synchronous ones can be expressed by asynchronous guarded actions.

Based on all these definitions, it is now possible to understand how a synchronous interpreter works. For the synchronous guarded actions, $P \subseteq S$ and t the number of clock cycles to run, the synchronous interpreter can be seen in Figure 1.1. Though it looks intimidating at the first look the interpreter does nothing more than read all inputs for a clock cycle, check if some delayed actions must be executed and computing the current reaction by calling the function *ComputeReaction*. After the current set of values for all variables and its delayed actions has been updated , this whole process is repeated. *ComputeReaction* on the other hand is more complicated. It s primary task is to execute all guarded actions until the computed set of values does not change any more. Furthermore is has to check if write-conflicts arise, since not every set of guarded actions must form a "program". Any Quartz program can now be translated to a set of guarded actions and executed by this interpreter.

However, this will only work, if we assume a synchronous environment, that features a global clock. On the contrary this interpreter is doomed to fail for any asynchronous environment, as there is no global clock. As asynchrony neglects timing behaviours, all incoming values for a component must not belong to the same synchronous reaction. The problem can be solved by embed-

function INTERPRETQUARTZSYNCHRONOUSLY(P,t)
  for x $\in$ Var(P) **do**
    ϵ_{def} = false;
  end for
  Del := {};
  $\epsilon_{pre} := \epsilon_{def}$;
  repeat
    ϵ_{in} := readInputs(); ▷ read all inputs
    for x $\in$ Del **do**
      ϵ_{in}(x) := ϵ_{pre}(x); ▷ execute delayed assignments
    end for
    (ϵ, D_{del}) := COMPUTEREACTION(P,ϵ_{in},ϵ_{def});
    for x $\in$ Var(P) $\wedge$ type(x) == 'mem' **do** ▷ update environment for reaction to absence
      ϵ_{def}(x) := ϵ(x);
    end for
    $\epsilon_{pre} := \epsilon$;
    t := t-1;
  until (t $\neq$ 0) ▷ For t $<$ 0, infinite behaviour
end function

function COMPUTEREACTION(P,ϵ,ϵ_{def})
  repeat
    Del := $\emptyset$;
    $\epsilon_{pre} := \epsilon$;
    for x $\in$ wrVar(P) **do** ▷ Reaction to Absence
      if $\forall \gamma \in P_x . \neg [grd(\gamma)]_{\epsilon_{pre}} \wedge \epsilon(x) = \bot$ **then**
        ϵ(x) := ϵ_{pre};
      end if
    end for
    for $\gamma \in$ P **do**
      if $[grd(\gamma)]_{\epsilon_{pre}} \wedge ([act(\gamma)]_{\epsilon_{pre}} \in \mathbb{B})$ **then** ▷ check if action can be executed
        if (wrVar(γ) $\neq \emptyset$) **then** ▷ check if instantaneous assignment
          if $(\epsilon(wrVar(\gamma))) \in \mathbb{B})$ **then** ▷ check if write conflict
            $\epsilon(wrVar(\gamma)) := \top$;
          else
            $\epsilon(wrVar(\gamma)) := act(\gamma)]_{\epsilon_{pre}}$;
          end if
        else
          if (Del $\cap$ nwrVar(γ) != $\emptyset$) **then** ▷ check if write conflict
            Del := $\top$;
          else
            Del := Del $\cup$ nwrVar(γ);
          end if
        end if
      end if
    end for
  until $(\epsilon \neq \epsilon_{pre})$
  if $\exists$ x $\in$ Var(P). $\epsilon(x) \in \{\bot, \top\} \vee$ Del = $\top$ **then** ▷ check for a valid behaviour
    fail;
  end if
  return (ϵ,Del)
end function

Figure 3.3: A synchronous interpreter for synchronous guarded actions

ding each component into a wrapper. This wrapper receives now all incoming variable values and decides when to trigger the synchronous component. The decision to trigger same reaction is thereby solely based on all variable values. Notice that the absence of a variable can no longer be interfered. If the wrapper decides to trigger the synchronous component, it must therefore read all required variable values and generate absence values, for those not available yet. But when does a wrapper trigger its synchronous component ? The answer is, the wrapper has to check, if there exists a reaction ϑ, that assign each variable its current value or absent and for ϑ it holds

$$\begin{array}{l}\forall x \in wrVar(P).\\ (\exists \alpha \in sgrdPa(P_x).[\alpha]_\vartheta = true)\vee\\ (\forall \alpha_i \in grdPa(P_x).Var(\alpha_i) \cap \vartheta = \emptyset)\vee\\ (\forall \alpha_i \in grdPa(P_x).(Var(\alpha_i) \cap \vartheta = Var(\alpha_i))\end{array}$$

Thus for write variable x it must hold that one clause of a guarded action writing x is satisfied by the reaction. If all clauses of all guarded actions writing x are false or its guard variables are all absent, the reaction is also valid. Either of these cases, satisfied for *all* write variables in P, means ϑ is a valid reaction and can be triggered. Of course it would be possible to define different conditions under which a wrapper can trigger its component, but it has shown that restricting these criteria too much leads to the result that only very few synchronous systems fulfill the criteria of endo-/isochrony. As this is not desirable, I opted for these conditions, leading to a fair amount of endo- and isochronous systems. Implementing then a interpreter that combines both a wrapper and a synchronous interpreter looks like Figure 2.2.

The interpreter works as sketched in the previous paragraph. For each received input value, it checks, whether a valid reaction can be constructed or not. Therefore it checks for all write variables, if some guarded action, writing this variable, can be executed. According to given truth tables, this is the case if one guard clause evaluates to true and the action evaluates to any boolean value. If so, the current environment is updated and all the variables needed for evaluating this guarded action are marked. After the environment does not change anymore, the interpreter first checks for eventual write conflicts. Two enabled guarded actions, writing the same variable with different values, might

be the reason for a write conflict and cause the interpreter to exit. Moreover the interpreter checks, that for all write variables, having no guard evaluated to true, all their guard either evaluate to false, such that all used guard variables are actually present. Or that all guards evaluate to $\perp$ and all guard variables are absent. For all guards evaluated to true, the guard variables are additionally marked. Having come so far, the last step is to remove the marked inputs from the buffer, as they do form a valid reaction. The produced reaction can than be thought as a synchronous reaction, triggered by a wrapper in an asynchronous environment.

So far it was shown, that any given Quartz program can be encoded by guarded actions and then executed by using the asynchronous interpreter. As the asynchronous interpreter produces synchronous behaviours, the question arises, if those synchronous behaviours are in relation to the behaviours, produced in a synchronous environment. While the concept of endochrony ensures that these behaviours are almost equivalent, isochrony is used to reason about multiple components, which interact with each other.

3.3 Basic Definitions

Many of the following definitions serve as the basis for the upcoming criteria and the explanation of endo-/isochronous systems. By a large part they have been taken from either [11] or [18]. Some of them need to be slightly modified to ensure consistency :

Definition 1 (Partial order of values/reactions/behaviours [11, 18])**.** *For a set of input and output values* $\mathbb{B} \cup \{*, \perp, \top\}$ *two partial orders are defined, namely:*

1. $\forall x \in \mathbb{B} : * \leq \perp \leq x \leq \top$

2. $* \sqsubseteq \perp \wedge \forall x \in \mathbb{B} : * \sqsubseteq x$

The difference is, while "$\leq$ always induces a greatest lower bound $\wedge$ and lowest upper bound $\vee$, relation $\sqsubseteq$ might induce only a greatest lower bound $\sqcap$[2] and does not relate to the value $\top$. Both relations are justified. On the one hand the synchronous/asynchronous interpreter is bound to generate a

[2]Of course there can exist $\sqcup$, but this will not be always the case

function InterpretQuartzAsynchronously(P)
repeat
Wait for Input; ▷ for each arriving input the execution proceeds
I := $\emptyset$
ϵ_{in} := readInputs();
repeat
$\epsilon_{pre} = \epsilon_{in}$;
for x $\in$ wrVar(P) **do**
if $[grd(P_x)]_{\epsilon_{pre}}$ **then** ▷ check if some guard evaluates to true
for $\gamma \in P_x$ **do**
if $[grd(\gamma)]_{\epsilon_{pre}} \wedge [act(\gamma)]_{\epsilon_{pre}} \in \mathbb{B}$ **then**
if $(\epsilon_{pre}(wrVar(\gamma)) = \bot \oplus \epsilon_{pre}(wrVar(\gamma)) = [act(\gamma)]_{\epsilon_{pre}})$ **then**
▷ check for write conflict
$\epsilon_{in} := [act(\gamma)]_{\epsilon_{in}}$;
else
$\epsilon_{in} := \top$;
end if
I := I $\cup$ rdVar(γ); ▷ mark read variables
I := I $\cup \{x | \exists \alpha \in sgrdPa(\gamma).[\alpha]_{\epsilon_{pre}} = true \wedge x \in Var(\alpha)\}$;
▷ mark variables, used in guard evaluated to true
end if
end for
end if
end for
until $\epsilon_{in} \neq \epsilon_{pre}$
if $\exists x \in Var(P).\epsilon_{in}(x) = \top$ **then** ▷ check if write conflict occurred
fail;
end if
for x $\in$ wrVar(P) **do**
if $([grd(P_x)]_{\epsilon_{in}} = \bot \wedge \exists x \in grdVar(P_x).\epsilon_{in}(x) \neq \bot)$ **then**
▷ check if one variable is present, although guards evaluate to absent
I = $\emptyset$;
end if
if $(\neg[grd(P_x)]_{\epsilon_{in}} \wedge \exists x \in grdVar(P_x).\epsilon_{in}(x) = \bot)$ **then**
▷ check if one variable is absent, although guards evaluate to false
I = $\emptyset$;
end if
end for
$(\epsilon_{out}, \epsilon_{in})$ = RemoveInputs(ϵ_{in},I); ▷ remove used values from Buffer
until (true)
end function

function RemoveInputs(ϵ_{in},I)
for x $\in$ Var(P) **do**
$\epsilon_{out}(x) := \bot$;
end for
for x $\in$ Var(P) **do**
if x $\in$ I **then**
$\epsilon_{out}(x) := \epsilon_{in}(x)$;
$\epsilon_{in}(x) := \bot$;
end if
end for
return $(\epsilon_{out}, \epsilon_{in})$; ▷ $\epsilon_{out} \equiv$ valid reaction, $\epsilon_{in} \equiv$ current buffer values
end function

Figure 3.4: An asynchronous interpreter for asynchronous guarded actions

reaction starting from absence values, but might encounter $\top$ due to write conflicts, on the other hand an ordinary "valid" reaction does not feature the value $\top$ at all, since it is defined in terms of its signature.

Extending both relations to reactions, is done variable-wise:
For reaction r_1, r_2 it holds
$r_1 \leq r_2 :\Leftrightarrow \forall x \in V : r_1(x) \leq r_2(x)$. and
$r_1 \sqsubseteq r_2 :\Leftrightarrow \forall x \in V : r_1(x) \sqsubseteq r_2(x)$.
Relation $\leq$ is also the reason why the interpreter, computing some reaction, actually terminates. Since the computed values do form a finite lattice, the computation must be continuous and therefore terminate with some fixpoint. As the relations for reactions are defined in terms of the variable relations, the same holds for the greatest lower and lowest upper bounds.

Finally the relations can be extended to behaviours, such that for p_1, p_2:
$p_1 \leq p_2 :\Leftrightarrow \forall t \in \mathbb{B} : r_1^t(x) \leq r_2^t(x)$. and
$p_1 \sqsubseteq p_2 :\Leftrightarrow \forall t \in \mathbb{B} : r_1^t(x) \sqsubseteq r_2^t(x)$.
and "t" denoting the reaction at some clock tick t.
As done before, greatest lower bound and lowest upper bounds are defined if every reaction defines it.
Two behaviours p_1, p_2 are then called *synchronizable* iff $\exists p_1 \sqcup p_2$.

Definition 2 (Clock of some variable/reaction [18]). *For a set of guarded actions $P \subseteq S$, a variable $x \in V$ and some reaction $r \in Rct(P)$. The clock of reaction is defined as*

$$cl(r) := \begin{cases} 0 & if\ \forall x \in grdVar(r).r(x) = \bot \\ 1 & otherwise \end{cases}$$

The clock of a reaction defines whether any guarded action of P can be fulfilled by this reaction. If there are only absent values to be read, no guarded action can evaluate to anything different than absent. Otherwise the clock of a variable for some reaction r is defined as:

$$cl(r, x) := \begin{cases} 0 & if\ r(x) = \bot \\ 1 & otherwise \end{cases}$$

It simply signals the occurrence of an value different to absent in the current reaction.

Definition 3 (Projection on reaction/behaviour [18]). *For any given $V' \subseteq V$*

and a reaction r, the projection of r to V', namely $r'_{|V}$ is defined as:

$$\forall v \in V'.r'_{|V}(v) := \begin{cases} r(v) & v \in V' \\ * & otherwise \end{cases}$$

The extension to behaviours is done similarly.
For $p := (r^t)_{t\in\mathbb{N}} \in Bhv(P)$ and $V' \subseteq V$ the projection of p to V', namely $p'_{|V}$ is defined as:

$$p'_{|V'} := (r^t_{V'})_{t\in\mathbb{N}} \wedge \forall t.r^t_{V'}(v) := \begin{cases} r^t(v) & v \in V' \\ * & otherwise \end{cases}$$

Definition 4 (Desynchronization [11])**.** *The desynchronization of a behaviour $p \in$ Bhv(P) aims at taking from p, for each variable, only the boolean values and ignoring any different values from the set $\{*, \perp, \top\}$.*
It is defined the following way:

$$flows(p) := \begin{cases} flows(p_1) \cdot flows(p_2) & \text{if } p := p_1 \cdot p_2 \\ fl(p) & otherwise \end{cases} \wedge$$

$$\forall v \in V : fl(r)(v) := \begin{cases} r(v) & \text{if } r(v) \in \mathbb{B} \\ o & otherwise \end{cases}$$

where $o \equiv$ the empty word
For a given set of behaviours P' $\subseteq$ P it is defined as:

$$flows(P') := \bigcup_{p_i \in P'} flows(p_i)$$

For example the desynchronization of behaviour $p(x) := 1 \cdot 1 \cdot \perp \cdot 0 \cdot \perp \cdot \perp \cdot 1 \ldots$ yields $flows(p) = 1 \cdot 1 \cdot 0 \cdot 1 \ldots$.

Definition 5 (Synchronous/Asynchronous Composition [18])**.** *Given two sets of guarded actions P, Q over variables $\mathcal{S}, \mathcal{T}$ and corresponding behaviours P' $\subseteq$ Bhv(P) and Q' $\subseteq$ Bhv(Q), the synchronous and asynchronous composition is defined as follows:*

- *P'$\|$Q' $:= \{\mathcal{E} \mid \exists p' \in P' \exists q' \in Q'.\mathcal{E} := p' \sqcup q'\}$*
- *P'$\|_{fl}$Q' $:= \{\mathcal{E} \mid \exists p' \in P' \exists q' \in Q'.\mathcal{E} := flows(p') \sqcup flows(q')\}$*

The synchronous composition tries to combine synchronous behaviours by looking if all the values of all shared variables agree. However the asynchronous

For p_1 and p_2 defined the following way:

$$p_1 = \begin{array}{r|ccc} p_1(x) & 1 & 1 & \bot \\ p_1(y) & 0 & 0 & \bot \end{array} \qquad p_2 = \begin{array}{r|ccc} p_1(y) & 0 & \bot & 0 \\ p_1(z) & 0 & \bot & 0 \end{array}$$

$$\Rightarrow p_1 \| p_2 = \begin{array}{r|cccc} p_1(x) & 1 & \bot & 1 & \bot \\ p_1(y) & 0 & \bot & 0 & \bot \\ p_1(z) & 0 & \bot & 0 & \bot \end{array} \quad \wedge \quad p_1 \|_{fl} p_2 = \begin{array}{r|cc} p_1(x) & 1 & 1 \\ p_1(y) & 0 & 0 \\ p_1(z) & 0 & 0 \end{array}$$

Figure 3.5: Synchronous and asynchronous composition of behaviours p_1, p_2.

composition does not regard synchronous behaviours, but only the set of desynchronized behaviours. Thus the value $\bot$ is ignored and only the sequence of boolean values for each variable is considered. It follows, that Bhv(P$\|$Q) = Bhv(P)$\|$Bhv(Q), as well as Bhv(P$\|_{fl}$Q) = Bhv(P)$\|_{fl}$Bhv(Q). Yet the more common notation of P$\|_{fl}$Q is flows(P)//flows(Q), which is reasonable, when looking at the definitions.
In Figure 3.5 a rather small example is given to illustrate how synchronous and asynchronous composition works.

Definition 6 (Stretching Function [18])**.** *A function $f : \mathbb{B} \rightarrow \mathbb{B}$ is called a stretching function, if the following holds:*

- $\forall t \in \mathbb{N}. t \leq f(t)$
- $\forall t_1, t_2 \in \mathbb{N}. t_1 \leq t_2 \rightarrow f(t_1) \leq f(t_2)$

A stretching function maps points of time, given by t, to later points of time and does this in a strictly monotonous way.

Definition 7 (Stretch and Flow Orders and Equivalences [18](Def. 7.38))**.** *For behaviours ξ and ρ, we define the following relations: ξ is a stretching of a behaviour ρ, written as $\rho -_{cl} \xi$ if there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that the following holds:*

- $\rho \dashv_{cl} \xi :\Leftrightarrow \exists f. \forall x \in V. \left(\begin{array}{c} Stretch(f) \wedge \\ (\forall t \in \mathbb{N}.[x]_\rho(f(t)) = [x]_\varrho(t)) \wedge \\ (\forall t \in \mathbb{N}.(\neg \exists t \in \mathbb{N}.t = f(t)) \rightarrow [x]_\varrho(t) = \bot) \end{array} \right)$

- $\rho \dashv_{fl} \xi :\Leftrightarrow \forall x \in V.\exists f. \begin{pmatrix} Stretch(f)\wedge \\ (\forall t \in \mathbb{N}.[x]_\rho(f(t)) = [x]_\varrho(t))\wedge \\ (\forall t \in \mathbb{N}.(\neg\exists t \in \mathbb{N}.t = f(t)) \rightarrow [x]_\varrho(t) = \bot) \end{pmatrix}$
- $\rho \approx_{cl} \xi :\Leftrightarrow \exists \varrho.\varrho \dashv_{cl} \rho \wedge \varrho \dashv_{cl} \xi$
- $\rho \approx_{fl} \xi :\Leftrightarrow \exists \varrho.\varrho \dashv_{fl} \rho \wedge \varrho \dashv_{fl} \xi$

Relations $\dashv_{cl}$ and $\dashv_{fl}$ are called stretch and flow orders, while $\approx_{cl}$ and $\approx_{fl}$ form stretch and flow equivalences. The idea of any stretch order is, that for given behaviours ρ and ξ, ρ can transformed into ξ by simply inserting reactions that do only consist of absence as a value, also called stuttering reactions. Flow orders on the other hand do hold, if ρ can be stretched variable-wise, by inserting absence values, to match the behaviour of ξ. Flow and clock equivalences between ρ and ξ simply require the existence of some behaviour that can be stretched by flow or by clock to match ρ respectively ξ. As flow and stretch relations form partial orders, flow and clock equivalences do even form equivalence relations[3]. In Figure 3.3 three behaviours, forming a certain order, are displayed. While p_1, p_2 are stretch ordered, as p_2 can be constructed from p_1 by adding stuttering reactions, p_1, p_3 are not stretch ordered but flow ordered, since only the stream of variable values equals, if all absence values are neglected. The very same holds for p_2, p_3, as they are not *stretch ordered, but flow ordered. Removing from p_1, p_2 and p_3 all absence values, it shows that all behaviours are not alone flow ordered, but also flow equivalent. In addition p_1 and p_2 are even clock equivalent, as the removal of all stuttering reactions yields equivalent behaviours.*

3.4 Endochrony

3.4.1 Definition

Definition 8 (Endochronous System [18](Def. 7.46)). *Any synchronous process P, respectively the set of guarded Actions, is called endochronous if for all $\rho_1, \rho_2 \in$ Bhv(P) with $\rho_1 \approx_{fl}^{in} \rho_2$, also $\rho_1 \approx_{cl} \rho_2$ holds.*

[3] For a proof see [18][Lemma 7.40/Lemma 7.42]

$p_1(x)$	1	$\perp$	0	1
$p_1(y)$	1	$\perp$	1	1
$p_1(z)$	1	$\perp$	0	1

$p_2(x)$	1	$\perp$	$\perp$	0	$\perp$	1
$p_2(y)$	1	$\perp$	$\perp$	1	$\perp$	1
$p_2(z)$	1	$\perp$	$\perp$	0	$\perp$	1

$p_3(x)$	1	$\perp$	0	1	$\perp$
$p_3(y)$	$\perp$	1	$\perp$	1	1
$p_3(z)$	1	$\perp$	$\perp$	0	1

Figure 3.6: It holds: $p_1 \dashv_{cl} p_2$, $p_1 \dashv_{fl} p_3$ and $p_2 \dashv_{fl} p_3$

Endochrony effectively states that an endochronous process is able to uniquely resynchronize any flow-equivalent behaviours. Thinking of an asynchronous environment a synchronous process can only be embedded by the help of a wrapper, that receives all incoming values and triggers the synchronous process. How to trigger a synchronous process is a tough question, since all synchronization by means of absence values has lost its meaning. It it therefore very likely that the synchronous process might be triggered with an entirely different set of values compared to a synchronous environment. However, if the synchronous process at hand is endochronous, the wrapper has always a way to determine the correct selection of values, such that the produced reaction will be equal to the one produced by a synchronous environment. As this wrapper will be in our case the proposed asynchronous interpreter, it must be clear that all criteria for endo-/isochrony are always bound to its execution scheme.

This can be seen by looking at Figure 3.7, which shows two different ways of implementing the boolean OR-function. The first guarded action C_1 produces behaviours p and p_1. Though both behaviours are input flow equivalent, they are not clock equivalent, as there exists no unique resynchronization for flow equivalent behaviours. This is due to C_1 and the execution of guarded action in asynchronous environments. Given C_1 it is obvious, that both clauses a or b are sufficient to trigger the execution of C_1. Reactions reading both variable values can therefore be separated and the resynchronization relies mostly on the environment, which contradicts the concept of endochrony. Yet, C_2 is able to resynchronize each input stream. What is difference between C_1 and C_2, as they both implement the same boolean function ? The answer is, for C_2

The single guarded action C_1: $(a \vee b) \Rightarrow x = 1$ allows behaviours:

$$\mathrm{p} = \begin{array}{r|ccc} \hline p(a) & 1 & \bot & 0 \\ p(b) & 0 & \bot & 1 \\ p(x) & 1 & \bot & 1 \\ \hline \end{array} \quad \wedge \quad p_1 = \begin{array}{r|ccc} \hline p_1(a) & 1 & 0 & \bot \\ p_1(b) & \bot & 0 & 1 \\ p_1(x) & 1 & \bot & 1 \\ \hline \end{array}$$

whereas C_2: $(a \wedge \neg b) \vee (\neg a \wedge b) \vee (a \wedge b) \Rightarrow x = 1$ only allows:

$$\mathrm{p} = \begin{array}{r|ccc} \hline p(a) & 1 & \bot & 0 \\ p(b) & 0 & \bot & 1 \\ p(x) & 1 & \bot & 1 \end{array} \quad \wedge \quad p_2 = \begin{array}{r|cc} \hline p_2(a) & 1 & 0 \\ p_2(b) & 0 & 1 \\ p_2(x) & 1 & 1 \end{array}$$

Figure 3.7: For p_1, p_2 and p_3 being flow equivalent, it holds:
$p \approx_{cl} p_2$ but $p \not\approx_{cl} p_1$, hence C_1 is *not* endochronous and C_2 is.

to be triggered, always both input values have to be read. In consequence, the reception of just one input value is not sufficient as it is for C_1. It seems as the form of a boolean function decides whether a component is considered endochronous or not.

Lemma 1. *The following statements (1) and (2) are equivalent.*

(1) $\forall x_1, x_2 \in wrVar(P).\forall \alpha_1 \in sgrdPa(P_{x_1}).\forall \alpha_2 \in sgrdPa(P_{x_2}).$
$Var(\alpha_1) \cap Var(\alpha_2) \neq \emptyset$

(2) $\forall x_1, x_2 \in wrVar(P) \forall \vartheta \in Rct(P).$
$cl(\vartheta_{|Var(P_{x1})}) = cl(\vartheta_{|Var(P_{x2})})$

The lemma states that regarding the asynchronous execution scheme, all guards will be either triggered simultaneously or no guard will be triggered at all. This is due to the fact that the clauses of all guards always share some variable and cannot be triggered alone.

Proof. "$\Rightarrow$ " $(1) \rightarrow (2) \Leftrightarrow \neg(2) \rightarrow \neg(1)$
By precondition $\exists x_1, x_2 \in wrVar(P) \exists \vartheta \in Rct(P).cl(\vartheta_{|Var(P_{x1})} \neq cl(\vartheta_{|Var(P_{x1})})$.
Think of $\vartheta_{|Var(P_{x1}}$ as ϑ_1 and $\vartheta_{|Var(P_{x2}}$ as ϑ_2.

Since Val(cl) $= \mathbb{B}$ and $cl(\vartheta_1) \neq cl(\vartheta_2)$ assume w.l.o.g. $cl(\vartheta_1) = 1 \wedge cl(\vartheta_2) = 0$. For $cl(\vartheta_2) = 0$ this implies $\forall x \in grdVar(P_{x_2}).\vartheta_2(x) = \bot$. As $cl(\vartheta_1) = 1$ there are two possible cases:

(i) $[grd(P_{x1})]_{\vartheta_1} = true \rightarrow \exists \gamma_1 \in P_{x1}.\exists \alpha_1 \in sgrdPa(\gamma_1).[\alpha_1]_\vartheta = true$.
But with $\forall x \in grdVar(P_{x2}).\vartheta_2(x) = \bot$ then $\forall \gamma_2 \in P_{x2}.\forall \alpha_2 \in sgrdPa(\gamma_2)$. $Var(\alpha_1) \cap Var(\alpha_2) = \emptyset$, otherwise ϑ_2 could not have the value $\bot$ for all guard variables

(ii) $[grd(P_{x1})]_{\vartheta_1} = false \rightarrow \forall \alpha_1 \in grdPa(P_{x1}).[\alpha_1]_\vartheta = false$.
But with $\forall x \in grdVar(P_{x2}).\vartheta_2(x) = \bot$ then $\forall \gamma_2 \in P_{x2}.\forall \alpha_2 \in sgrdPa(\gamma_2)$. $Var(\alpha_1) \cap Var(\alpha_2) = \emptyset$, otherwise ϑ_2 could not have the value $\bot$ for all guard variables

In both cases $\exists \alpha_1 \alpha_2.Var(\alpha_1) \cap Var(\alpha_2) = \emptyset$ and so $\neg(1)$ holds.

"$\Leftarrow$" $(2) \rightarrow (1) \Leftrightarrow \neg(1) \rightarrow \neg(2)$
$\exists x_1, x_2 \in wrVar(P).\exists \alpha_1 \in sgrdPa(P_{x1}).\exists \alpha_2 \in sgrdPa(P_{x2})$.
$Var(\alpha_1) \cap Var(\alpha_2) = \emptyset$.
This implies
$\exists \vartheta \in Rct(P).\vartheta := \vartheta_1 \sqcup \vartheta_2 \wedge [\alpha_1]_\vartheta = true \wedge \forall x \in grdVar(P_{x2}).\vartheta(x) = \bot$.
As $[\alpha_1]_\vartheta = true$ implies $cl(\vartheta_1) = 1$ and $\forall x \in grdVar(P_{x_2}).\vartheta(x) = \bot$ implies $cl(\vartheta_2) = 0$ then
$\exists x_1, x_2 \in wrVar(P).\exists \vartheta \in Rct(P).cl(\vartheta_{|Var(P_{x1})} = \vartheta_1) = cl(\vartheta_{|Var(P_{x1})} = \vartheta_2)$

□

Guarded actions like:

$$(a \wedge b) \Rightarrow x_1 = 1$$
$$(b \wedge c) \Rightarrow x_2 = 1$$

can only be fired simultaneously, as both guard clauses $(a \wedge b)$ and $(b \wedge c)$ share the variable b. In reference to the asynchronous execution model, no behaviour exists that can trigger just one guarded action, as this always means that the second guarded action is evaluated in way that either partially satisfies or partially unsatisfies the guard clause. As this contradicts the execution model, both guarded action have the same clock.

A latency insensitive component, composed of two endochronous guarded actions

$$a \Rightarrow x_1 = 1$$
$$b \Rightarrow x_2 = 1$$

yields behaviours: $p_1 = \begin{array}{c|cc} p_1(a) & 1 & \bot \\ p_1(b) & \bot & 1 \\ p_1(x_1) & 1 & \bot \\ p_1(x_2) & \bot & 1 \end{array} \quad \wedge \quad p_2 = \begin{array}{c|cc} p_2(a) & \bot & 1 \\ p_2(b) & \bot & 1 \\ p_2(x_1) & \bot & 1 \\ p_2(x_2) & \bot & 1 \end{array}$

Figure 3.8: It holds: From $p_1 \approx_{fl}^{in} p_2$ results $p_1 \approx_{fl}^{out} p_1$, hence C_1 is latency insensitive but not endochronous.

3.4.2 Latency Insensitivity

Definition 9 (Latency insensitive systems [18](Def. 7.49))**.** *A synchronous system is latency insensitive if for all $p_1, p_2 \in Bhv(P)$ with $p_1 \approx_{fl}^{in} p_2$, we also have $p_1 \approx_{fl}^{out} p_2$.*

Compared to endochrony, latency insensitivity does not provide the exact timing behaviour, such that flow equivalent behaviours are resynchronized for every point of time. Nevertheless, it states that the set of produced output values will be, for flow equivalent behaviours, always the same. As this respects the logical order of values, it perfectly matches the requirement of an asynchronous environment, that guarantees only the ordering of values. Often those processes are composed of multiple processes, that do not interfere with each other. While every process may be endochronous and uniquely resynchronizes its incoming values, the outputs might be produced to different points of time as compared to an synchronous environment. An example of this case can be seen in Figure 3.8.

Lemma 2. *If P and Q are latency insensitive systems, then also $P \parallel Q$ are latency insensitive.*
Hence the composition of latency insenstive processes yields a latency insenstive process.

Proof. precondition:
P latency insensitive over variables $S \subseteq V \wedge S := S_{in} \cup S_{out}$

$\Leftrightarrow (\forall p_1, p_2 \in Bhv(P).p_1 \approx^{S_{in}}_{flow} p_2 \rightarrow p_1 \approx^{S_{out}}_{flow} p_2)\wedge$
Q latency insensitive over variables $T \subseteq V \wedge T := T_{in} \cup T_{out}$
$\Leftrightarrow (\forall q_1, q_2 \in Bhv(Q).q_1 \approx^{T_{in}}_{flow} q_2 \rightarrow q_1 \approx^{T_{out}}_{flow} q_2)\wedge$
to show:
$(\forall b_1, b_2 \in Bhv(P||Q).b_1 \approx^{B_{in}}_{fl} b_2 \rightarrow b_1 \approx^{B_{out}}_{fl} b_2)\wedge$
$B_{in} := (S_{in} \cup T_{in})\backslash((S_{out} \cap T_{in}) \cup (T_{out} \cap S_{in}))\wedge$
$B_{out} := (S_{out} \cup T_{out})\backslash((S_{out} \cap T_{in}) \cup (T_{out} \cap S_{in}))$
it holds: Bhv(P||Q) := Bhv(P) || Bhv(Q) $\rightarrow \forall b \in Bhv(P||Q)[\exists b_p \in Bhv(P)\exists b_q \in Bhv(Q).\forall v \in B \cap S.b(v) = b_p(v) \wedge \forall v \in B \cap T.b(v) = b_q(v)]$
For given $b_1, b_2 \in Bhv(P||Q).b_1 \approx^{B_{in}}_{fl} b_2$ it holds
$\exists b_{1p}, b_{2p} \in Bhv(P) \wedge \exists b_{1q}, b_{2q} \in Bhv(Q).$
$\forall v \in B \cap S.b_1(v) = b_{1p}(v) \wedge b_2(v) = b_{2p}(v)\wedge$
$\forall v \in B \cap T.b_1(v) = b_{1q}(v) \wedge b_2(v) = b_{2q}(v)\wedge$
(*)$\forall v \in S \cap T.b_{1p}(v) = b_{1q}(v) \wedge b_{2p}(v) = b_{2q}(v)$
(**)Because of $b_1 \approx^{B_{in}}_{fl} b_2 \rightarrow b_{1p} \approx^{S_{in}}_{fl} b_{2p} \xrightarrow{\text{P lat.ins.}} b_{1p} \approx^{S_{out}}_{fl} b_{2p}$
(***)Because of$b_1 \approx^{B_{in}}_{fl} b_2 \rightarrow b_{1q} \approx^{T_{in}}_{fl} b_{2q} \xrightarrow{\text{Q lat.ins.}} b_{1q} \approx^{T_{out}}_{fl} b_{2q}$
With $b = b_p||b_q$ it holds:
$b_1 \approx^{B_{in}}_{fl} b_2 \Leftrightarrow b_{1p}||b_{1q} \approx^{B_{in}}_{fl} b_{2p}||b_{2q} \xrightarrow{(*)-(***)} b_{1p}||b_{1q} \approx^{B_{out}}_{fl} b_{2p}||b_{2q} \Leftrightarrow b_1 \approx^{B_{out}}_{fl} b_2$.

□

3.4.3 Criteria

Criterion 1. *Given $P \subseteq S$ a set of guarded actions, P is considered to be endochronous iff*

1. *P fulfills Statement (1) of Lemma 1* $\wedge$
2. $\forall r \in Rct(P).\forall_{x \in wrVar}.\forall \gamma_1, \gamma_2 \in P_x.\forall \alpha_1 \in grdPa(\gamma_1).\forall \alpha_2 \in grdPa(\gamma_2).$ $[\alpha_1]_r = [\alpha_2]_r = true \rightarrow \alpha_1 = \alpha_2$

The most important statement of Criterion 1 concerns processes writing just one variable. For any such process P it holds, that P must be endochronous, iff from the set of all guard clauses always just one clause is fulfilled by a valid reaction. This allows to uniquely resynchronize any inputs received in an asynchronous environment, the main characteristic of endochrony. In the case that

a reaction exists, that fulfills more than just one clause, P features behaviours that disagree with the definition of endochrony.
Finally Statement (1) of Lemma 1 is needed to prove, that also processes writing multiple guard variables can be endochronous, as long as each set of guarded actions writing just one variable is endochronous and the guards of all guarded actions of the process, share the same clock.

Proof. "$\Leftarrow$"
First assume P $\subseteq$ S. |wrVar(P)| = 1.
It is known that (1) and (2) hold by precondition.
W.l.o.g. assume wrVar(P) = {x}.
Think of arbitrary behaviours $p_1, p_2 \in Bhv(P) \wedge p_1 \approx_{fl}^{in} p_2$ and p_1 as well as p_2 not only composed of stuttering reactions. Behaviours composed only of stuttering reactions, do not need to be considered as they must be endochronous.
Now there $\exists t_1, t_2 \in \mathbb{N}. \exists y \in grdVar(P)$.
$p_1^{t_1}(y) = p_2^{t_2}(y) \wedge p_1^{t_1}(y) \neq \bot \wedge \forall t < t_1. p_1^t(y) = \bot \wedge \forall t < t_2. p_2^t(y) = \bot$.
Then t_1 respectively t_2 can be regarded as the first point of time in p_1/p_2 where some variable holds a value, as a consequence all preceding reactions must be therefore stuttering reactions.

If $y \in grdVar(P_x) \wedge p_1^{t_1}(x) = \bot$ then because of the execution scheme
$\forall y' \in grdVar(P_x). p_1^{t_1}(y') \neq \bot \wedge \forall v \in Var(P) \backslash grdVar(p_1). p_1^{t_1}(v) = \bot$.
Since $p_1 \approx_{fl} p_2$ holds by precondition, also $p_2^{t_2}(x) = \bot$ must hold.
But this implies $[p_x]_{p_2^{t_2}} = false$ and that is only allowed to happen if
$\forall y \in grdVar(p_x). p_2^{t_2}(y) \neq \bot \wedge \forall v \in Var(P) \backslash grdVar(P_x). p_2^{t_2} = \bot$.
Thus $\forall v \in V. p_1^{t_1}(v) = p_2^{t_2}$ holds.

If $y \in grdVar(P_x) \wedge p_1^{t_1} \neq \bot$ it is clear $\exists \alpha_2. [\alpha_2]_{p_1^{t_1}} = true$.
Now if $p_2^{t_2}(x) = \bot$ holds it suggest again that all guard variables must have a value. Therefore and because y is the first value to be, holds
$\forall v \in Var(\alpha_2). p_1^{t_1}(v) = p_2^{t_2}(v)$.
However, then $p_1^{t_1}(x) = p_2^{t_2}(x)$ holds in contrast to the assumption. Hence $p_2^{t_2}(x) \neq \bot$ and $\exists \gamma_1, \gamma_2 \in P. \exists \alpha_1 \in sgrdPa(\gamma_1). \exists \alpha_2 \in sgrdPa(\gamma_2)$.
$[\alpha_1]_{p_1^{t_1}} = [\alpha_2]_{p_2^{t_2}} = true$.
The case $p_1^{t_1}(x) \neq p_2^{t_2}(x)$ is not possible. Since this would mean

$\exists\vartheta. = p_1^{t_1} \sqcup p_2^{t_2}$ with $\vartheta(x) = true \wedge \vartheta(x) = false$.
No behaviour can carry two values for one variable. Thus $p_1^{t_1}(x) = p_2^{t_2}(x)$ holds. In addition it is known that $p_1^{t_1} \approx_{fl}^{in} p_2^{t_2}$ holds.
If now $\exists v \in Var(P_x).p_1^{t_1}(v) \neq p_2^{t_2}(v)$ then
$\exists\vartheta := p_1^{t_1} \sqcup p_2^{t_2}$ and $\exists\alpha_k, \alpha_j.[\alpha_k]_\vartheta = [\alpha_j]_\vartheta = true$ but $\alpha_k \neq \alpha_j$, since $p_1^{t_1}$ and $p_2^{t_2}$ differ in at least one variable.
This is a contradiction to precondition (1) and so $\forall v \in V.p_1^{t_1}(v) = p_2^{t_2}(v)$ holds.

Defining $T_k := \{t' \mid t' \in \mathbb{N} \wedge \exists v \in V.p_k^{t'}(v) \neq \bot\}$, it is now an easy thing to show that
$\forall t_1 \in T_1.\exists t_2 \in T.\forall v \in V.p_1^{t_1}(v) = p_2^{t_2}(v) \wedge$
$|\{t \mid t \in T_1 \wedge t \leq t_1\}| = |\{t \mid t \in T_2 \wedge t \leq t_2\}|$.
It was shown that his holds for $t_1 = min(T_1)$ and $t_2 = min(T_2)$ with
$|\{t|t \in T_1 \wedge t \leq t_1\}| = |\{t|t \in T_2 \wedge t \leq t_2\}|$. The trick is to remove t_1/t_2 from their respective sets and to consider only $T_1' := T_1 \backslash \{t_1\}$ and $T_2' := T_2 \backslash \{t_2\}$. For those sets the very same reasoning can be used as before, since they have new minima. The cut from T_1 respectively T_2 is perfectly valid since behaviours are defined in terms of reactions. As this procedure can be repeated over and over again, every new step yields longer prefixes $p_1' \sqsubseteq p_1 \wedge p_2' \sqsubseteq p_2$, for which it is known that $p_1' \approx_{cl} p_2'$ holds. Although p_1, p_2 might be infinite behaviours, the clock equivalence is proofed for any prefix, making $p_1 \approx_{cl} p_2$ and therefore P endochronous.

Until now all guarded actions had to agree on the one write variable. But if there are two or more variables to write, things change. This is were Lemma 1 can be used. The statements of Lemma 1 state that all guards P' have to have the same clock. In other words if one guard is triggered the other ones are triggered and if one guard is not triggered the reaction must be a stuttering reaction. The reasoning for P and |wrVar(P)| > 1 is as follows. Any P can be split up into $P := \bigcup_{x \in wrVar(P)} P_x := \{\gamma \mid \gamma \in P \wedge wrVar(\gamma) = \{x\}\}$.
As every subset P_x has only one write variable and also complies with the preconditions iff P did, for any $p_{x1}, p_{x2} \in Bhv(P_x).p_{x1} \approx_{fl}^{in} p_{x2} \rightarrow p_{x1} \approx_{cl} p_{x2}$, as it was shown above. Yet $\forall p_1/p_2 \in Bhv(P).p_1/p_2 := \sqcup_{x \in wrVar(P)} p_x \in P_x$. If $p_1 \approx_{cl}^{in} p_2$ holds and all P_x are endochronous and because of Statement 1, the clocks of all guards are the same, $p_1 \approx_{cl} p_2$ must hold. Consequently P is

endochronous even for multiple write variables given that Statement 1 holds.

"$\Rightarrow$"
Instead of showing P endochronous $\Rightarrow$ (1) and (2), it is proved, that P is not endochronous $\Leftarrow \neg(1) \vee \neg(2)$. For that, a case distinction is made
case 1:
(1) Statement 1 does not apply. This is equivalent to the existence of at least two guarded actions featuring different write variables such that
$\exists x_1, x_2 \in wrVar(P) \exists \alpha_1 \in sgrdPa(P_{x1}) \exists \alpha_2 \in sgrdPa(P_{x2})$.
$Var(\alpha_1) \cap Var(\alpha_2) = \emptyset$, because Statement 1 does not hold.
Based on that there
$\exists \vartheta_1, \vartheta_2 \in Rct(P)$.
$[\alpha_1]_{\vartheta_1} = true \wedge \forall x \in grdVar(P_{x1}).\vartheta_1(x) = \perp \wedge$
$[\alpha_2]_{\vartheta_2} = true \wedge \forall x \in grdVar(P_{x2}).\vartheta_2(x) = \perp$
$\rightarrow \vartheta_1(x_1) \neq \perp \wedge \vartheta_1(x_2) = \perp \wedge \vartheta_2(x_2) \neq \perp \wedge \vartheta_2(x_1) = \perp$.
In consequence $\exists \vartheta_1 \cdot \vartheta_2 \cdot \epsilon, \vartheta_2 \cdot \vartheta_1 \cdot \epsilon \in$ Bhv(P). Though it holds $\vartheta_1 \cdot \vartheta_2 \cdot \epsilon \approx_{fl}^{in} \vartheta_2 \cdot \vartheta_1 \cdot \epsilon$ it does not hold $\vartheta_1 \cdot \vartheta_2 \cdot \epsilon \approx_{fl}^{out} \vartheta_2 \cdot \vartheta_1 \cdot \epsilon$ and therefore $\vartheta_1 \cdot \vartheta_2 \cdot \epsilon \not\approx_{cl} \vartheta_2 \cdot \vartheta_1 \cdot \epsilon$, so that P not endochronous[4].

case 2:
$\neg(2)$ means
$\exists \vartheta \in Rct(P).\exists x \in wrVar(P).\exists \gamma_1, \gamma_2 \in P_x.\exists \alpha_1 \in sgrdPa(\gamma_1).\exists \alpha_2 \in sgrdPa(\gamma_2)$.
$[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true \wedge \alpha_1 \neq \alpha_2$.
As $\alpha_1 \neq \alpha_2$ but $[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true$ there exists at least one variable x_1 such that w.l.o.g. $x_1 \in Var(\alpha_1) \wedge x_1 \notin Var(\alpha_2)$.
This suggests that there exists $\vartheta'(x) := \begin{cases} \perp \text{ , if } x \in Var(\alpha_1) \wedge x \notin Var(\alpha_2) \\ \vartheta(x) \text{ , otherwise} \end{cases}$
that is still a valid behaviour. Considering now $\vartheta \cdot \vartheta', \vartheta' \cdot \vartheta \cdot \epsilon \in Bhv(P)$, $\vartheta \cdot \vartheta' \cdot \epsilon \approx_{fl} \vartheta' \cdot \vartheta \cdot \epsilon$ holds but they are not clock equivalent because of x_1. Thus, $\vartheta \cdot \vartheta' \cdot \epsilon \not\approx_{cl} \vartheta' \cdot \vartheta \cdot \epsilon$ holds, and P cannot be endochronous[5].

case 3:

[4]For an example to this case, see Figure 3.8.
[5]For an example to this case, see Figure 3.9

A guarded action like

$$a \vee (a \wedge b) \Rightarrow x = 1$$

allows behaviours:

$$p_1 = \begin{array}{c|cc} p_1(a) & 1 & 1 \\ p_1(b) & 1 & \perp \\ p_1(x) & 1 & 1 \end{array} \quad \wedge \quad p_2 = \begin{array}{c|cc} p_2(a) & 1 & 1 \\ p_2(b) & \perp & 1 \\ p_2(x) & 1 & 1 \end{array}$$

Figure 3.9: For p_1, p_2 it holds: $p_1 \approx_{fl} p_2$ but obviously $p_1 \not\approx_{cl} p_2$

case 1) *and* case 2) are fulfilled, resulting in P being not endochronous.
As a result of cases 1) - 3) it holds $\neg(1) \vee \neg(2) \Rightarrow$ P not endochronous $\Leftrightarrow$
P endochronous $\Rightarrow (1) \wedge (2)$. □

It seems that Statement 1 has the role to coordinate all the synchronous guards to ensure the endochrony of the encompassing program. What happens if this coordination does not take place ? Is there still any synchronization left such that the effects of an asynchronous environment can be handled ? As Criteria 2 shows, something is actually left, and it is simply latency insensitivity.

Criterion 2. *For proving the latency insensitivity of any given process P, the process must be partitioned into sets of guarded actions, writing the same variable. For each of those sets it might be the case, that the value assigned to its write variable is for each reaction true or for each reaction false. Then the set of guarded actions is almost latency insensitive except for one possibility, that must be ruled out.*
The other case is more likely to occur. It states, the written variable of a set of guarded actions can carry different boolean values. As consequence the constraints on these guarded actions must be tighter than compared to the first case. The case distinction is made clear by separating the criterion into two parts, applicable in either case.

For given $P \subseteq S$, P is latency insensitive iff
$\forall x \in wrVar(P)$ *it holds*

1. $\forall \vartheta_1, \vartheta_2 \in Rct(P) \wedge \vartheta_1(x), \vartheta_2(x) \in \mathbb{B}.$
   $\vartheta_1(x) = \vartheta_2(x) \rightarrow \forall \alpha_1, \alpha_2 \in sgrdPa(P_x).Var(\alpha_1) \cap Var(\alpha_2) \neq \emptyset$

2. $\exists \vartheta_1, \vartheta_2 \in Rct(P) \wedge \vartheta_1(x), \vartheta_2(x) \in \mathbb{B}.$
   $\vartheta_1(x) = \vartheta_2(x) \rightarrow 2(a) \wedge 2(b) \wedge 2(c)$
   Where

   (a) $\forall \gamma_1, \gamma_2 \in P_{true}.\forall \alpha_1 \in sgrdPa(\gamma_1).\forall \alpha_2 \in sgrdPa(\gamma_2).$
   $Var(\alpha_1) \cap Var(\alpha_2) \neq \emptyset$

   (b) $\forall \gamma_1, \gamma_2 \in P_{false}.\forall \alpha_1 \in sgrdPa(\gamma_1).\forall \alpha_2 \in sgrdPa(\gamma_2).$
   $Var(\alpha_1) \cap Var(\alpha_2) \neq \emptyset$

   (c) $\forall \phi_1 \in P_{true}.\forall \phi_2 \in P_{false}.\forall \vartheta \in Rct(P).\forall \alpha_1 \in grdPa(\phi_1).\forall \alpha_2 \in grdPa(\phi_2).$
   $[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true \rightarrow \alpha_1 = \alpha_2$

   $\wedge\ P_{b \in \{true, false\}} := \{\gamma \mid \gamma \in P_x \wedge \exists \vartheta \in Rct(P).\vartheta(x) = b\}$

Proof. "$\Leftarrow$"
Assume |wrVar(P)| = 1 $\wedge$ wrVar(P) = {x}, then two cases might occur:
case 1:
$\forall \vartheta_1, \vartheta_2 \in Rct(P) \wedge \vartheta_1(x), \vartheta_2(x) \in \mathbb{B}.\vartheta_1(x) = \vartheta_2(x)$
Thus x always carries either the value true or false.
If now (3) $\forall \vartheta \in Rct(P) \forall \gamma_1, \gamma_2 \in P \forall \alpha_1 \in sgrdPa(\gamma_1) \forall \alpha_2 \in sgrdPa(\gamma_2).$
$[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true \rightarrow \alpha_1 = \alpha_2$
holds, P is even endochronous, since Criteria 1 holds (Statement 1/2 of Lemma 1 are negligible, because x is just one write variable).
This implies that P must be latency insensitive.
If !(3) but 1(a) still holds, then $\exists \gamma_i, \gamma_j \in P_x \exists \alpha_i \in sgrdPa(\gamma_i) \exists \alpha_j \in sgrdPa(\gamma_j).$
$Var(\alpha_i) \subseteq Var(\alpha_j) \wedge Var(\alpha_j) \not\subseteq Var(\alpha_i).$
For P to be not latency insensitive $\exists r_1, \ldots, r_n \in Rct(P) \exists r'_1, \ldots, r'_{n+1} \in Rct(P).$
$r_1, \ldots, r_n, r'_1, \ldots, r'_{n+1} \neq \bot \wedge Var(r'_i) \subseteq Var(r_i) \wedge$
$r_1 \cdots r_n \approx^{in}_{fl} r'_1, \cdots, r'_n, r'_{n+1} \wedge \sum_{i=1}^{n} cl(r_i, x) = \sum_{i=1}^{n+1} cl(r'_i, x).$
The idea is, since there is only one value for x, P can only be not latency

insensitive, if it manages to create for flow-equivalent input behaviours a varying amount of output values. Hence, there must exist a smallest sequence of reactions that generates, when executed, an amount of x values that differs by one value, depending on the way the input stream is read. The sequence of r_i''s always reads at most the same amount of values. Now
$X'_{last} := \{(x_i, r_i(x)) | x \in Var(r_i) \backslash Var(r_i') \land \exists i \forall j > i.r_j(x) = \bot; i, j \in \{1, \ldots, n\}\}$
and
$X_{last} := \{(x_i, r_i(x)) | x \in Var(r_i) \land \exists i \forall j > i.r_j(x) = \bot; i, j \in \{1, \ldots, n\}\}$
can be defined, X_{last} assigning each variable its last value different from $\bot$ and X'_{last} grouping all variables values not used in any reaction until r'_{n+1}. Obviously X'_{last} fulfills r'_{n+1}, but since $X'_{last} \subseteq X_{last}$, X_{last} also fulfills r'_{n+1}. In addition X_{last} fulfills r_n and therefore r'_n. Because r'_n and r'_{n+1} are fulfilled by X_{last} it must hold that $Var(r'_n) \cap Var(r'_{n+1}) = \emptyset$. Otherwise there would exist one variable, that would have been already read by r'_n, though it was needed in r'_{n+1}.
As $Var(r'_n) \cap Var(r'_{n+1}) = \emptyset$ contradicts precondition 1(a), the assumption of P being not latency insensitive must be wrong and P is therefore latency insensitive.

case 2:
$\exists \vartheta_1, \vartheta_2 \in Rct(P) \land \vartheta_1, \vartheta_2 \in \mathbb{B}.\vartheta_1(x) \neq \vartheta_2(x)$.
In addition assume that P is not latency insensitive. Write variable x is can now carry any boolean value. This allows for some classification of all guarded actions that assign x the value true or false. It is plausible to assume that some guarded actions will be in both sets. Taking a look at $P_{true} \backslash P_{false}$ and $P_{false} \backslash P_{true}$ it shows that both sets are latency insensitive, since all guarded actions write just one value and share variables for each clause. Hence, the conditions of case 1 are fulfilled. This constrains the reasons why P could theoretically be not latency insensitive, while this is not due to any $\gamma \in P_{true} \backslash P_{false} \cup P_{false} \backslash P_{true}$ as 2(c) states that no reaction belonging to one set interferes with the other set. Thus the reason for not being latency insensitive is not caused by a varying amount of output values. Instead, it must be caused by those $\gamma \in P_{true} \cap P_{false}$, such that for flow-equivalent input behaviours, some output values are simply swapped. This lead to the following observations. There must

$\exists r_1, \ldots, r_n \in Rct(P).\exists r'_1, \ldots, r'_n \in Rct(P).$
$r_1, \ldots, r_n, r'_1, \ldots, r'_n \neq \bot \wedge$
$r_n(x) \neq r'_n(x) \wedge \forall i < n.r_i(x) = r'_i(x) \wedge$
$r_1 \cdots r_n \approx^{in}_{fl} r'_1, \cdots, r'_n, r'_{n+1}.$
Since they are input flow equivalent, it must hold that
$\forall y \in Var(P).r_n(y) \neq \bot \wedge r'_n(y) \neq \bot \rightarrow r_n(y) = r'_n(y).$
In consequence $\exists \vartheta \in Rct(P).\vartheta := r_n \sqcup r'_n$ and
$\exists \gamma_1, \gamma_2 \in P_{true} \cap P_{false}.\exists \alpha_1 \in sgrdPa(\gamma_1) \exists \gamma_2 \in sgrdPa(\gamma_2).$
$[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true \wedge \alpha_1 \neq \alpha_2.$
But $\gamma_1, \gamma_2 \in P_{true} \cap P_{false}$ is equivalent to $\gamma_1, \gamma_2 \in P_{false} \wedge \gamma_1, \gamma_2 \in P_{false}$, such that the former result leads to a contradiction with 2(c). As $\gamma \in P_{true} \cap P_{false}$ does not prevent P from being latency insensitive, the initial assumption must be wrong and P latency insensitive.

Until now it was assumed that P had only one write variable. With the results so far, it easy to show that having multiple write variables does not influence the latency insensitivity of P. The set of guarded actions P can also be written as $P := \bigcup_{x \in wrVar(P)} P_x$. As each of those P_x has only one write variable, either case 1 or case 2 holds, making all P_x latency insensitive. But for P being composed of all those P_x and the preservation of latency insensitivity for a composition, it is necessary for P to be latency insensitive as well.

"$\Rightarrow$" Show reverse direction for a given $x \in wrVar(P)$
<u>case 1:</u> $\neg 1$(a) $\rightarrow$ P not latency insensitive
$\exists \gamma \in P_x \exists \alpha_1, \alpha_2 \in sgrdPa(\gamma).Var(\alpha_1) \cap Var(\alpha_2) = \emptyset.$
This implies $\exists r \cdot \epsilon_1 \in Bhv(P) \wedge \exists r_1 \cdot r_2 \cdot \epsilon_1 \in Bhv(P).$
$r \cdot \epsilon_1 \approx^{in}_{fl} r_1 \cdot r_2 \cdot \epsilon_1 \wedge [\alpha_1]_r = [\alpha_1]_{r_1} = true \wedge [\alpha_2]_r = [\alpha_2]_{r_2} = true.$
From this follows that $cl(r, x) + 1 = cl(r_1, x) + cl(r_2, x)$. Thus one output value more has been produced and since both behaviours only differ in r and r_1, r_2, such that $r \cdot \epsilon_1 \not\approx^{out}_{fl} r_1 \cdot r_2 \cdot \epsilon_1$ holds. Therefore P is not latency insensitive.

<u>case 2:</u> $\neg 2(a) \vee \neg 2(b) \vee \neg 2(c) \rightarrow$ P not latency insensitive.
Negation of 2(a) or 2(b) leads to
$\exists \gamma_1, \gamma_2 \in P_{true/false}.\exists \alpha_1 \in sgrdPa(\gamma_1).\exists \alpha_2 \in sgrdPa(\gamma_2).$

$Var(\alpha_1) \cap Var(\alpha_2) = \emptyset$.
If $\gamma_1, \gamma_2 \in P_{true} \vee \gamma_1, \gamma_2 \in P_{false}$ and
$\exists \vartheta \in Rct(P).[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true$ then case 1 applies, since both γ_1, γ_2 assign x the same value.
If $\gamma_1, \gamma_2 \in P_{true} \vee \gamma_1, \gamma_2 \in P_{false}$ and
$\forall \vartheta_1 \in \{\vartheta \mid \vartheta \in Rct(P) \wedge [\alpha_1]_\vartheta = true\}$.
$\forall \vartheta_2 \in \{\vartheta \mid \vartheta \in Rct(P) \wedge [\alpha_2]_\vartheta = true\}$.
$\vartheta_1(x) \neq \vartheta_2(x)$,
then $\exists \vartheta_1 \in Rct(P).[\alpha_1]_{\vartheta_1} = true \wedge [\alpha_2]_\vartheta = \perp$ and
$\exists \vartheta_2 \in Rct(P).[\alpha_2]_{\vartheta_2} = true \wedge [\alpha_1]_\vartheta = \perp$ and it holds for
$\vartheta_1 \cdot \vartheta_2 \cdot \epsilon_1, \vartheta_2 \cdot \vartheta_1 \cdot \epsilon_1 \in Bhv(P)$ that $\vartheta_1 \cdot \vartheta_2 \cdot \epsilon_1 \approx^{in}_{fl} \vartheta_2 \cdot \vartheta_1 \cdot \epsilon_1$.
But as $\vartheta_2(x) \neq \vartheta_1(x)$ this implies $\vartheta_1 \cdot \vartheta_2 \cdot \epsilon_1 \not\approx^{out}_{fl} \vartheta_2 \cdot \vartheta_1 \cdot \epsilon_1$.
As a result P is not latency insensitive.

The negation of 2(c) is equivalent to
$\exists \phi_1 \in P_{true}.\exists \phi_2 \in P_{false}.\exists \vartheta \in Rct(P).\exists \alpha_1 \in sgrdPa(\phi_1).\exists \alpha_2 \in sgrdPa(\phi_2)$.
$[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true \wedge \alpha_1 \neq \alpha_2$.
If $[\alpha_1]_\vartheta = [\alpha_2]_\vartheta = true \wedge \alpha_1 \neq \alpha_2$ then
$\exists \vartheta \cdot \epsilon_1, \vartheta_1 \cdot \vartheta_2 \cdot \epsilon_1 \in Bhv(P).\vartheta \cdot \epsilon_1 \approx^{in}_{fl} \vartheta_1 \cdot \vartheta_2 \cdot \epsilon_1$ but with
$cl(\vartheta, x) + 1 = cl(\vartheta_1, x) + cl(\vartheta_2, x)$ the behaviours cannot be output flow equivalent. Thus $\vartheta \cdot \epsilon_1 \not\approx^{out}_{fl} \vartheta_1 \cdot \vartheta_2 \cdot \epsilon_1$ holds and P is not latency insensitive.
Again very much the same arguments as in case 1 or previous prove schemes.
To sum up, if $\neg 2(a)$ or $\neg 2(b)$ or $\neg 2(c)$ holds, P is not latency insensitive.

□

3.5 Isochrony

3.5.1 Definition

Definition 10 (Isochronous Composition of Systems [18](Def. 7.53))**.** *The parallel composition of synchronous processes P and Q is isochronous if the following holds:*

$$flows(Bhv(P \| Q)) = flows(Bhv(P)) \|_{fl} flows(Bhv(Q))$$

Analogously the sets P' $\subseteq$ P and Q' $\subseteq$ Q are isochronous if it holds that

$$flows(P'\|Q') = flows(P')\|_{fl}flows(Q')$$

The desynchronization of a compositional synchronous system means that the system is split up into various synchronous components connected by an asynchronous environment. As the asynchrony might lead to unexpected behaviours of the decomposed system, isochrony ensures that each of those behaviours might be also produced by the compositional system. One might say while endochrony makes sure that each asynchronous behaviour can be deterministically processed, isochrony ensures that the communication between components is not altered by the asynchronous environment.

This can be seen by a simple example. For process P_1 composed of guarded actions

$$a \Rightarrow x_1 = 1$$
$$\neg a \Rightarrow x_2 = 1$$

and P_2 composed of

$$x_1 \wedge x_2 \Rightarrow y = 1$$

there exists no synchronous behaviour, different than a stuttering reaction, for the composition $P_1\|P_2$. This is the caused by the fact, that P_1 can only write either x_1 or x_2, but never both variables at once and P_2 must read x_1 and x_2 or no variable at all, for each reaction. As these two behaviours always conflict with each other, the composition does solely consist of the stuttering reaction.

However, things look different for an asynchronous environment. In an asynchronous environment x_1 and x_2 are still written by P_1 as it would be the case for a synchronous environment. But now the values might arrive at P_2 at any point of time. Thus there exists a behaviour where both x_1 and x_2 are received by P_2 at the same point of time and a reaction different than the stuttering reaction is produced. In consequence, the criterion of isochrony cannot be fulfilled, as the asynchronous composition of $P_1\|P_2$ does produce behaviours that are otherwise not generated by a synchronous composition. In Figure 3.10 such a behaviour is displayed for the processes P_1 and P_2.

$$p_1 = \begin{array}{r|cc} p(a) & 1 & 0 \\ p(x_1) & 1 & \bot \\ p(x_2) & \bot & 1 \\ \hline \end{array} \qquad p_2 = \begin{array}{r|cc} p_2(x_1) & 1 & \bot \\ p_2(x_2) & 1 & \bot \\ p_2(y) & 1 & \bot \\ \hline \end{array}$$

$$\Rightarrow \text{Flows}(p_1 \| p_2) = \emptyset$$

$$\Rightarrow \text{Flows}(p_1) \parallel \text{Flows}(p_2) = \begin{array}{r|c} \hline p_{12}(a) & 1 \\ p_{12}(x_1) & 1 \\ p_{12}(x_2) & 1 \\ p_{12}(y) & 1 \\ \hline \end{array} \neq \emptyset$$

Figure 3.10: Though p_1 and p_2 do not make up for a synchronous composition, they can asynchronously be combined.

3.5.2 Criteria

Having defined the term isochrony, the question arises if there is a way to check for some composition of synchronous processes if those form a isochronous composition or not. The following criteria shows how this can be done.

Criterion 3. *For $P \subseteq S$ and a list of adjacent processes, i.e. they share communication, $S_{out} = \{S_1, \dots, S_n\} \wedge S_i \subseteq S$ such that $\forall S_i.\exists x \in wrVar(S_i).x \in (grdVar(P) \cup rdVar(P))$, $P \| S_i$ is considered isochronous if $\forall \vartheta \in Rct(P)$.*

(1) $(\exists \vartheta_{S_i} \in Rct(S_i).\forall x \in Cut_{PS_i}.\vartheta(x) = \vartheta_{S_i}(x)) \vee$

(2) $(\forall \vartheta_{S_i} \in Rct(S_i).\exists x \in Var(grd(P)).\vartheta(x) \neq \vartheta_{S_i}(x))$*;*

Where $Cut_{PS_i} := \{x \mid \exists \gamma_s \in S_i.\exists \gamma_p \in P.(x \in grdVar(\gamma_s) \cup rdVar(\gamma_s) \wedge x \in wrVar(\gamma_p)) \vee (x \in grdVar(\gamma_p) \cup rdVar(\gamma_p) \wedge x \in wrVar(\gamma_s))\}$

The essential meaning of this criterion is, that for each process P_1 that is connected to some process P_2, the composition of $P_1 \| P_2$ must be isochronous, if for each feasible reaction r_1 of P_1 a reaction r_2 of P_2 can be found, such that both reactions agree on all shared variables (the set $Cut_{P_1P_2}$).

Proof. The criterion for checking isochrony is separated into two parts. Part (2) states that for each chosen reaction, it might be the case that this reaction will never be invoked by the composition P$||S_i$. This might be caused by the fact that the set of guard variables of this reaction cannot be written by providing S_i with appropriate values. Hence, the chosen reaction is of no relevance for the decision whether P$||S_i$ is isochronous or not. Things are different for part (1). For any possible reaction ϑ to occur in a composition of P$||S_i$, there must exist some reaction ϑ_{S_i} of S_i such that ϑ and ϑ_{S_i} agree on all shared variables, defined by Cut_{PS_i}.Given two adjacent processes and two flow-equivalent behaviours for each process, it must hold that the synchronous behaviour of the receiving process is composed of reactions. For each of those reactions it holds by part (1) that there exists some reaction of the sending process that writes exactly those values read by the receiver. As the produced set of reactions for the sender forms a synchronous behaviour, this behaviour and the behaviour of the receiver can be composed. Thus the composed behaviour is flow equivalent to both viewed behaviours and therefore P$||S_i$ must be isochronous. □

The former criteria essentially states, that two synchronous processes can always be checked for isochrony. The nice thing about this is, that it can already be done during compile time. Although the complexity of this model checking might be considerable, the compiler has to do it only once and can afterwards guarantee, that the program at hand is isochronous.

Criterion 4. *For $P_1, P_2, P_3 \subseteq S$ it holds*
if $Var(P_2) \cap Var(P_3) = \emptyset \wedge P_1\|P_2, P_1\|P_3$ isochronous $\wedge P_1$ endochronous
$\rightarrow P_1\|P_2\|P_3$ isochronous

Given processes P_1, P_2 and P_3, the criterion states, that the endochrony of P_1, the isochrony of compositions $P_1\|P_2$,$P_1\|P3$ and no shared variables between P_2 and P_3 are sufficient to reason about the isochrony of the composition of all three processes. Key to the isochrony of the overall composition is the endochrony of P_1, as shown in the following proof.

Proof. By precondition it is known that $P_1\|P_2 \wedge P_1\|P_3$ are isochronous.
Since $Var(P_2) \cap Var(P_3) = \emptyset$, this implies

$\forall p_2 \in Bhv(P_2) \wedge \forall p_3 \in Bhv(P_3).\ \exists p_2 \sqcup p_3 \in Bhv(P_2 \| P_3), P_2 \| P_3$
must be isochronous too.
Now $P_1 \| P_2 \| P_3$ is by definition considered isochronous iff
flows(Bhv($P_1 \| P_2 \| P_3$)) = flows(Bhv(P_1))‖flows(Bhv(P_2))‖flows(Bhv(P_3))

i) flows(Bhv($P_1 \| P_2 \| P_3$)) $\subseteq$ flows(Bhv(P_1))‖flows(Bhv(P_2))‖flows(Bhv(P_3))
$\forall p \in$ Bhv($P_1 \| P_2 \| P_3$)) it holds
$\exists p_1 \in Bhv(P_1) \wedge \exists p_2 \in Bhv(P_2) \wedge \exists p_3 \in Bhv(P_3).p := p_1 \| p_2 \| p_3$.
With that flows(p) = flows($p_1 \| p_2 \| p_3$) = flows(p_1)‖flows(p_2)‖flows(p_3) holds.

ii) flows(Bhv($P_1 \| P_2 \| P_3$)) $\supseteq$ flows(Bhv(P_1))‖flows(Bhv(P_2))‖flows(Bhv(P_3))
This is equivalent to showing that
$\forall p_1 \in Bhv(P_1) \forall p_2 \in Bhv(P_2) \forall p_3 \in Bhv(P_3).p_1 \approx_{fl} p_2 \wedge p_1 \approx_{fl} p_3$
$\rightarrow \exists \epsilon \in Bhv(P_1 \| P_2 \| P_3).\epsilon \approx_{fl} p_1 \wedge \epsilon \approx_{fl} p_2 \wedge \epsilon \approx_{fl} p_3$.
Given such p_1, p_2, p_3 then by using the commutativity of $||$
flows(Bhv(P_1))‖flows(Bhv(P_2))‖flows(Bhv(P_3)) = (flows(p_1)||flows(p_2))||(flows(p_1)||flows(p_3))
Since $P_1 || P_2 \wedge P_1 || P_3$ are isochronous by precondition
$\exists p_{12} \in$ Bhv($P_1 || P_2$). flaws(p_{12}) = flaws(p_1)||flaws(p_2) $\wedge$
$\exists p_{13} \in$ Bhv($P_1 || P_3$). flaws(p_{13}) = flaws(p_1)||flaws(p_3)
$\rightarrow$ (flows(p_1)||flows(p_2))||(flows(p_1)||flows(p_3)) = flows(p_{12})||flows(p_{13})
Given that P_1 is endochronous and $p_{12|Var(P_1)} \approx_{fl} p_{13|Var(P_1)}$
follows $p_{12|Var(P_1)} \approx_{cl} p_{13|Var(P_1)}$.
As $Var(P_2) \cap Var(P_3)$ holds, it is trivial task to construct
$p_{123} \in Bvh(P_1 || P_2 || P_3).p_{123} \approx_{cl} p_{12} \wedge p_{123} \approx_{cl} p_{13}$, knowing P_1 is endochronous.
Thus p_{123} is the synchronous behaviour such that
$p_{123} \approx_{fl} p_1 \wedge p_{123} \approx_{fl} p_2 \wedge p_{123} \approx_{fl} p_3$ is fulfilled.

Showing both inclusions result in equality between the two sets flows(Bhv($P_1 \| P_2 \| P_3$)) and flows(Bhv(P_1))‖flows(Bhv(P_2))‖flows(Bhv(P_3)), proofing that under given conditions for P_1, P_2, P_3, the synchronous composition $P_1 || P_2 || P_3$ is isochronous.

□

Former criteria is especially astonishing, since in general the isochrony of parts of a composition does *not* imply the isochrony of the whole composition [18][Lemma 7.57ff.]. Given now certain constraints, it shows that once again some processes composed the right way are actually enough to make a

composition isochronous.

Criterion 5. *The composition of a network of synchronous processes* $P := \{P_1, \ldots, P_n\}$ *is to be isochronous iff*

1. $\forall P_i \in P.|wrVar(P_i)| = 1$
2. $\forall P_i, P_j \in P.(\exists x \in Cut_{P_iP_j} \rightarrow \forall_{k \neq j} P_k.Cut_{P_iP_k} = \emptyset)$
3. $\forall P' := (P_1, \ldots, P_k) \in P^k.\forall_{i \in \{1,\ldots,k\}} P_i.\exists x \in Cut_{P_iP_{i+1}} \wedge P_i \neq P_{i+1}$
   $\rightarrow \forall_{l,m \in \{1,\ldots,k\}} P_l, P_m.P_l \neq P_m$

Constraint (1) states that two processes are only allowed to share one variable. (2) defines that each process has at most one successor whereas (3) demands the network to be acyclic. As a result, each network of synchronous components build like a tree structure, must be isochronous.

Proof. Given a set of processes P := $\{P_1, \ldots, P_n\}$ fulfilling (1) to (3)
show: flows(Bhv($P_1 || \ldots || P_n$)) = flows(Bhv(P_1))$|| \ldots ||$flows(Bhv(P_n))

"$\Rightarrow$"
This inclusion is always valid, for thorough proof see criteria 4.

"$\Leftarrow$"
Assume $\exists p_1 \in Bhv(P_1), \ldots, p_n \in Bhv(P_n).flows(p_1)|| \ldots ||flows(p_n)$ is possible.
Then there must $\exists p_c \in Bhv(P_1|| \ldots ||P_n).p_c \approx_{fl} p_1 \wedge \ldots \wedge p_c \approx_{fl} p_n$.
Further assume P_r is the root process and $\exists P' := \{P_1, \ldots, P_k\} \subseteq P$.
$\forall_{i \in \{1,\ldots,k\}} P_i.Cut_{P_rP_i} = \{x_i\} \wedge \forall_{i \in \{1,\ldots,k\}} t_i = 0 \wedge t_r = 0$. Check if one of the following cases applies:
case 1:
$\forall_{i \in \{1,\ldots,k\}} i.p_r^{t_r}(x_i) = p_i^{t_i}(x_i)$
$\rightarrow p_r := p_r + 1 \wedge \forall_{i \in \{1,\ldots,k\}} i.p_i := p_i + 1$
case 2:
$\forall_{i \in \{1,\ldots,k\}} i.p_i^{t_i}(x_i) = \perp$
$\rightarrow p_r := stutter(p_r)$
case 3:
$\exists_{i,j \in \{1,\ldots,k\}} i, j.p_i^{t_i}(x_i) = \perp \wedge p_j^{t_j}(x_j) \neq \perp$

$\rightarrow \forall_{l \in \{1,\ldots,k\}} l.(p_l^{t_l}(x_l) \neq \bot \rightarrow p_l := stutter(p_l))$
case 4:
$\forall_{i \in \{1,\ldots,k\}} i.p_i^{t_i}(x_i) \neq \bot$
$\rightarrow \forall_{l \in \{1,\ldots,k\}} l.(p_l^{t_l}(x_l) \neq p_r^{t_r}(x_l) \rightarrow p_l := stutter(p_l))$
where $stutter(p) := p_\bot \cdot p$

Repeating this "checking" indefinitely will yield clock equivalent behaviours $p'_r, p'_1, \ldots, p'_k$ such that $p'_r||p'_1||\ldots||p'_k$ can be constructed. Since this composition might not include all processes, the whole procedure has to be repeated for all processes that share a variable with the root node. After that, the procedure is repeated for their children processes and so on. This is the typical breadth first search that is guaranteed to be complete for any tree structure. One word on the length of these behaviours. As most of the time only infinite behaviours will be considered, the first run of the algorithm will not terminate. Hence executing all these steps sequentially will have the result that all succeeding runs will not start, since they rely on the former runs. The solution to this problem is concurrency. After the first run has advanced in parsing both p_r and all p_i behaviours, the preliminary result can be used as a starting point for all succeeding runs of the algorithm. If a new result is available, since the parser has advanced, it is sufficient to cut the already used prefix and to use the resulting input for all succeeding runs. Results gained by all succeeding runs can then simply be concatenated, forming the final result. All of this is only feasible because all produced behaviours are clock-equivalent and do form partial-orders. □

To illustrate how the algorithm of Criteria 5 works, a network of synchronous processes is displayed in the upper half of Figure 3.11. The lower half of Figure 3.11 shows behaviours for each given process. It is important, that all behaviours are at least flow equivalent on the shared variables to another. The algorithm, used in the proof of Criteria 5, is now able to construct a compositional behaviour from these behaviours, that is clock equivalent to all given behaviours. To do so, it starts at the root process C_1 and tries to build a compositional behaviour for C_1 and all its child process, namely C_2 and C_3. In each step the algorithm makes the case distinction provided in the proof of Criteria 5 and checks which case holds. If one case is satisfied, its corresponding

changes are carried out. Notice that all of those changes are correct, as they do only add silent reactions to a behaviour. For building a compositional behaviour the algorithm has to repeat the procedure over and over. The first steps of this *computation* are shown in Figure 3.11, the transitions marked by the satisfied case.

Summarizing, Criteria 5 shows an interesting fact. If a network of synchronous processes does form a specific structure, given by criteria 3, it is guaranteed to be isochronous. It looks like isochrony is sometimes only a question of how to wire different synchronous processes so that a certain structure is created. Especially in times of exploding circuit sizes, these findings could be useful to some extent.

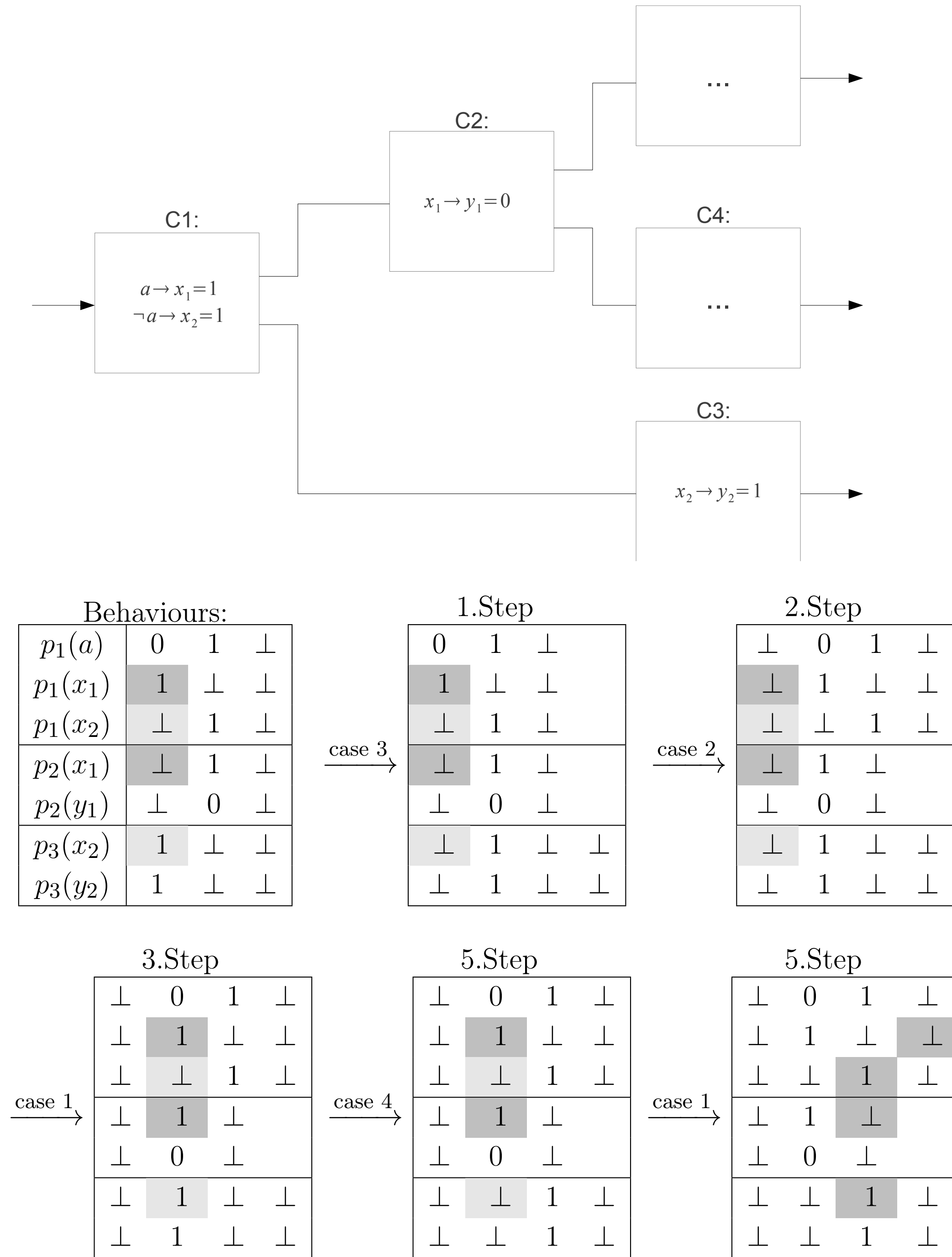

Figure 3.11: A synchronous network of processes and behaviours for processes C_1, C_2 and C_3, which are processed by the algorithm of Criteria 5

4 Summary

4.1 Conclusion

The main goal of this thesis was to work out, how synchronous guarded actions can be run in an asynchronous environment and the consequences this yields. While the definition of a asynchronous interpreter helped resolving the first issue, the criteria for classifying endo- or isochronous processes proofed to be complex and partially hard to grasp. Nevertheless presentable results have been achieved, as some of those criteria only rely on the main definitions of endo-/isochrony and are free of eventual influences by the interpreter.

As further elaborations on those topics might come, it would be interesting to see someone work out a few of the issues that have been hardly addressed in this thesis. To name a few, the feasibility of implementing an efficient asynchronous interpreter for synchronous guarded actions as well as coming up with an efficient checking algorithm for both endo- and isochrony, that can be run during compile time. For my own part, I hope that this thesis clarified some of the issues concerning synchronous languages and provided results useful in further research.

Bibliography

[1] B. Caillaud A. Beneviste and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation 1,2,3. *Inf. Comput.*, 163(1):125–171, November 2000.

[2] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[3] G. Berry. The constructive semantics of pure Esterel, July 1999.

[4] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[5] J. Brandt and K. Schneider. Static data-flow analysis of synchronous programs. In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, MEMOCODE'09, pages 161–170, Piscataway, NJ, USA, 2009. IEEE Press.

[6] D.M. Chapiro. *Globally-Asynchronous, Locally-Synchronous Systems*. PhD thesis, Department of Computer Science, Stanford, California, USA, 1984.

[7] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[8] M. Gemunde J. Brandt and K. Schneider. Desynchronizing synchronous programs by modes. *Application of Concurrency to System Design, International Conference on*, 0:32–41, 2009.

[9] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Application of Concurrency to System Design (ACSD)*, pages 48–57, Saint-Malo, France, 2005. IEEE Computer Society.

[10] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. IRISA Technical report PI 1670, Institut National de Recherche en Informatique et en Automatique (INRIA), 2005.

[11] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Application of Concurrency to System Design (ACSD)*, pages 67–76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[12] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design (FMSD)*, 28(2):111–130, 2006.

[13] D. Potop-Butucaru, R. de Simone, and Y. Sorel. From multi-clock constraints to multi-rate GALS executives. Research Report 6021, INRIA, Sophia Antipolis, France, November 2006.

[14] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In C.M. Kirsch and R. Wilhelm, editors, *Embedded Software (EMSOFT)*, pages 124–133, Salzburg, Austria, 2007. ACM.

[15] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. Research Report 6152, INRIA, Sophia Antipolis, France, March 2007.

[16] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In S. Chakraborty and N. Halbwachs, editors, *Embedded Software (EMSOFT)*, pages 147–156, Grenoble, France, 2009. ACM.

[17] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. In S. Edwards, R. Lorenz, and W. Vogler, editors, *Application of Concurrency to System Design (ACSD)*, pages 42–51, Augsburg, Germany, 2009. IEEE Computer Society.

[18] K. Schneider. The synchronous programming language quartz, 2009.