
Mike Gemünde

Clock Refinement in Imperative Synchronous Languages

vom Fachbereich Informatik der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Datum der wissenschaftlichen Aussprache	18. Oktober 2013
Dekan	Prof. Dr. Arnd Poetzsch-Heffter
Vorsitzender der Promotionskommission	Prof. Dr. Markus Nebel
Berichterstatter	Prof. Dr. Klaus Schneider Prof. Dr. Michael Mendler Prof. Dr. Sandeep Shukla

D 386

Danksagung

An dieser Stelle möchte ich allen danken, die zur Entstehung dieser Arbeit und dem Gelingen meiner Promotion beigetragen haben. Besonders hervorheben möchte ich dabei Herrn Prof. Dr. Klaus Schneider, der mir die Arbeit erst ermöglicht hat und mir stets mit offenem Ohr und Ratschlägen unterstützend zur Seite stand. Darüber hinaus möchte ich mich bei den weiteren Gutachtern, Herrn Prof. Dr. Michael Mendler und Herrn Prof. Dr. Sandeep Shukla, sowie bei dem Vorsitzenden der Promotionskommission, Herrn Prof. Dr. Markus Nebel, bedanken. Ferner bedanke ich mich bei meinen Kollegen für die netten Gespräche und neuen Denkanstöße. Nicht zuletzt geht besonderer Dank für die langjährige Unterstützung an meine Familie, ohne die ich nicht zu diesem Punkt gekommen wäre.

Oktober 2013, *Mike Gemünde*

Contents

Zusammenfassung	1
Abstract	3
1 Introduction	5
1.1 Motivation	5
1.2 Contribution	6
1.3 Outline	8
2 Related Work	9
2.1 Models of Parallel Computation	9
2.1.1 Synchronous Model	9
2.1.2 Asynchronous (Untimed) Model	18
2.1.3 Discrete Event Model	18
2.2 The Synchronous Language QUARTZ	19
2.2.1 Introductory Examples	19
2.2.2 Statements	22
2.2.3 Semantic Issues	25
2.2.4 Formal Semantics	30
2.2.5 Intermediate Representation	32
2.2.6 Compilation	34
2.2.7 Code Generation	37
2.2.8 AVEREST	42
3 Clock Refinement	45
3.1 Limitations of QUARTZ	45
3.2 Basic Idea of Clock Refinement	46
3.2.1 Steps, Variables and Assignments	49
3.2.2 Parallel Execution	51
3.2.3 Abortion and Suspension	52
3.2.4 Determinism	53
3.3 Constructivity vs. Logical Correctness	54
3.3.1 Sequential Execution of Substeps	55

3.3.2	Scheduling of Parallel Threads	56
3.3.3	Steps and Instants	59
3.4	Summary	60
4	Formal Semantics	63
4.1	Definitions	63
4.2	Overview	66
4.3	Transition Rules	67
4.3.1	General Form of the Rules	67
4.3.2	Basic Statements	68
4.3.3	Strong Preemption	73
4.3.4	Weak Preemption	76
4.4	Reaction Rules	82
4.4.1	General Form of the Rules	82
4.4.2	Basic Statements	83
4.4.3	Strong Preemption	87
4.4.4	Weak Preemption	88
4.5	Program Execution	89
4.5.1	Interpreter	89
4.5.2	Constructive Execution	96
4.6	Summary	97
5	Compilation	99
5.1	Extended Intermediate Format	99
5.1.1	General Idea	100
5.1.2	Labels and Clocks	101
5.1.3	Local Declarations	101
5.1.4	Complete Structure	103
5.2	Surface and Depth	104
5.3	Translation of Certain Statements	105
5.3.1	Control-Flow Graph	105
5.3.2	Parallel Threads	106
5.3.3	Loops and Local Declarations	108
5.3.4	Strong Preemption	110
5.3.5	Weak Preemption	111
5.4	Compilation Algorithm	113
5.4.1	Definitions	115
5.4.2	Compile Functions	117
5.5	Checking Constructive Abstractions	120
5.6	Summary	122
6	Hardware Synthesis	123
6.1	Overall Structure	124
6.2	Functional Part	125
6.2.1	Representation of Hardware	125
6.2.2	Translation of Control Flow	126

6.2.3	Translation of Data Flow	126
6.2.4	Optimizations for Data-Flow	129
6.3	Scheduler	134
6.4	Summary	135
7	Evaluation	137
7.1	Examples	137
7.1.1	JPEG Example	137
7.1.2	Experimental Results	148
7.2	Comparison with Related Work	150
7.2.1	ESTEREL	150
7.2.2	Multiclock ESTEREL	151
7.2.3	SIGNAL	151
7.2.4	LUSTRE	153
7.3	Summary	153
8	Conclusion	155
	References	157
	Curriculum Vitae	167

Zusammenfassung

Zur Entwicklung eingebetteter Systeme wurde bereits eine Fülle von Berechnungsmodellen und Sprachen vorgestellt. Durch die direkte Unterstützung spezieller Anforderungen eingebetteter Systeme eignen sich diese Sprachen besser als traditionelle sequentielle Programmiersprachen. Die direkte Unterstützung nebenläufiger Berechnungen oder die wiederkehrende Interaktion mit der Umgebung sind nur einige der Vorteile solcher Sprachen. Ein spezielles Beispiel sind synchrone Sprachen. Sie zeichnen sich dadurch aus, die Ausführung eines Systems in eine Sequenz logischer Schritte zu unterteilen. Ein Schritt folgt dabei der Vereinfachung, dass Ausgaben direkt nach der Verfügbarkeit der Eingaben berechnet sind. Diese Abstraktion ermöglicht die wohldefinierte deterministische Komposition im Allgemeinen, sowie das deterministische Abbrechen oder Anhalten von Berechnungen in imperativen synchronen Sprachen im Speziellen. Darüber hinaus erlauben diese Eigenschaften die Übersetzung von Programmen in Hardware und Software sowie die direkte Anwendung von formalen Verifikationstechniken wie beispielsweise die Modellprüfung.

Mit den Vorteilen imperativer synchroner Sprachen gehen auch einige Nachteile einher. Ein Effekt der von der parallelen Ausführung in dem synchronen Modell verursacht wird ist die Übersynchronisierung von Threads, welche auch dann jeden Schritt gemeinsam ausführen, wenn sie nicht kommunizieren. Diese Arbeit betrachtet die Erweiterung von imperativen synchronen Sprachen mit *Clock-Refinement*, welche es erlaubt zusätzlich zu den Berechnungsschritten weitere Abstraktionsebenen einzuführen. Ein Schritt kann dabei in kleinere logische Schritte einer neu deklarierten Clock unterteilt werden um unabhängige Berechnungen durchzuführen. Diese explizite Beschreibung kann anschließend von Compilern zur Analyse verwendet werden. Darüber hinaus bleibt auf jeder der neuen Abstraktionsebenen das synchrone Modell mit seinen Vorteilen erhalten.

Die Erweiterung wird in dieser Arbeit auf Basis der imperativen synchronen Sprache QUARTZ präsentiert. Es werden neue Anweisungen eingeführt, welche es erlauben neue Clocks als Verfeinerung bestehender Clocks zu definieren. Die Auswirkungen der neuen Mechanismen auf die bestehende Sprache werden untersucht und die Semantik der Erweiterung wird formal definiert. Darüber hinaus wird ein Übersetzungsalgorithmus vorgestellt, der es erlaubt Programme in ein Zwischenformat zu übersetzen. Ausgehend von diesem Zwischenformat wird außerdem die Übersetzung in Hardware beschrieben. Die Vorteile der Spracherweiterung werden anschließend in Beispielen evaluiert.

Abstract

An huge amount of computational models and programming languages have been proposed for the description of embedded systems. In contrast to traditional sequential programming languages, they cope directly with the requirements for embedded systems: direct support for concurrent computations and periodic interaction with the environment are only some of the features they offer. Synchronous languages are one class of languages for the development of embedded systems and they follow the fundamental principle that the execution is divided into a sequence of logical steps. Thereby, each step follows the simplification that the computation of the outputs is finished directly when the inputs are available. This rigorous abstraction leads to well-defined deterministic parallel composition in general, and to deterministic abortion and suspension in imperative synchronous languages in particular. These key features also allow to translate programs to hardware and software, and also formal verification techniques like model checking can be easily applied.

Besides the advantages of imperative synchronous languages, also some drawbacks can be listed. *Over-synchronization* is an effect being caused by parallel threads which have to synchronize for each execution step, even if they do not communicate, since the synchronization is implicitly forced by the control-flow. This thesis considers the idea of clock refinement to introduce several abstraction layers for communication and synchronization in addition to the existing single-clock abstraction. Thereby, clocks can be refined by several independent clocks so that a controlled amount of asynchrony between subsequent synchronization points can be exploited by compilers. The declarations of clocks form a tree, and clocks can be defined within the threads of the parallel statement, which allows one to do independent computations based on these clocks without synchronizing the threads. However, the synchronous abstraction is kept at each level of the abstraction.

Clock refinement is introduced in this thesis as an extension to the imperative synchronous language QUARTZ. Therefore, new program statements are introduced which allow to define a new clock as a refinement of an existing one and to finish a step based on a certain clock. Examples are considered to show the impact of the behavior of the new statements to the already existing statements, before the semantics of this extension is formally defined. Furthermore, the thesis presents a compile algorithm to translate programs to an intermediate format, and to translate the intermediate format to a hardware description. The advantages obtained by the new modeling feature are finally evaluated based on examples.

Chapter 1

Introduction

1.1 Motivation

Embedded systems found their way in many applications like cars, avionics, factory automation, and also in many other products used in everybody's daily life. Computing units first substituted only small parts like simple mechanics or analogous circuits to be more flexible or just to be able to produce cheaper products, but their complexity, quantity, and interaction hugely increased and there is still no end in sight. For example, nowadays high-end cars contain around 70 electronic control units (ECU), and the development costs of software and hardware seems to exceed 40% of total development costs, but drives most of the innovation. The ECUs are located everywhere in the car and have to communicate and interact as a distributed system. However, not only in this application, but also on a single chip, multiple heterogeneous computational units can be found in form of a system on a chip (SoC) or multiprocessor system on a chip (MPSoC). Furthermore, embedded systems have to meet special requirements like e. g. resource consumption, real-time constraints, or reliability. Most of the people, can for example, get over an interrupted call with a cell phone, but a failure or delay of car breaks is simply not acceptable.

Parallel programming in general is challenging, since often traditional languages are reused for this purpose which introduces non-determinism that often cannot be properly handled by programmers. Adding statements for parallel computation to used sequential languages like C makes program analysis and verification difficult [Berr89], because concurrency and communication are implicit and hidden from the programmer. For this reason, also Lee [Lee06] advocated for languages that model basic deterministic parallel computations, but carefully introduce non-determinism where it is useful, but in a way that it will not be harmful. For heterogeneous embedded systems that have to meet special constraints, the development process will not become easier. Fortunately, *model-based design* became widely-used for the design of embedded systems, where different models have been developed [LeSa98, GiLL99, Jant04] which differ in the notion of when and how computations are performed, and how communication between parallel parts is established.

Furthermore, a lot of programming languages following the abstractions of the models have been developed in the past, and they can be translated to software, hardware, or even to both [BeKi00, Edwa00, Edwa05b, BrGS10]. Further results have been achieved in comparing the models based on more general frameworks [LeSa98, Jant04], or also in integrating different

models for combined simulation or code generation [BGSS11, BGSS12]. Model-based design flows start usually with an abstract description of the final system and this description is then refined by either using other models or by refining the description based on the already used model. Additionally, the increasing complexity of systems and the need for handling more complex problems like image processing or other signal processing tasks introduces the need of more models that are reliable and able to handle these problems in an efficient way.

1.2 Contribution

The contribution of this work is an extension [GeBS10, GeBS13] to the imperative synchronous language QUARTZ [Schn09] which follows the synchronous model of computation [BeBe91, BCEH03]. The model simplifies programming by abstracting from communication and computation delays and considers *ideal* systems that produce their outputs immediately when the inputs arrive; this ability is also referred to as *perfect synchrony* [Halb93]. This idealized view to the system lead to well-defined deterministic concurrent computations and lets the programmer focus on the actual functional behavior of the system. Moreover, it provides a model to describe reactive systems since it allows periodic reactions.

In each reaction, all (available) inputs are given to the system, all values are computed, and the outputs are provided. Furthermore, the internal state for the following reaction is updated. In this way, all signals only have one value per reaction leading to deterministic parallelism by synchronizing the reactions. As the synchronous abstraction simplifies programming on the one hand, it challenges compilers and tools on the other hand. Thereby, also the QUARTZ compiler has to deal with the well-known issues for imperative synchronous languages, namely *causality* and *schizophrenia*. Causality addresses the question which action happens *before* another one. Schizophrenia occurs if in one reaction, the scope of a locally declared variable is left *and* entered, so that two (or more) instances exists at the same time.

The synchronous model is used in control-flow based (imperative) languages like QUARTZ [Schn09], ESTEREL [BeCo85], and its graphical variants Statecharts [HaPn85] and SyncCharts [Andr03], in data-flow languages like LUSTRE [LeMe87], and also in polychronous specifications like SIGNAL [GaGB87, GGBM91], and its graphical variant MRICDF [JPSS09]. The latter ones also belong to data-flow languages but have a more declarative style of programming. The languages can be translated to hardware and software [Edwa00, Berr91], or the modules can be also distributed [GiNi03, BrGS09].

Parallel execution is inherent in the data-flow languages where several nodes communicate via communication channels and the execution is synchronized by the availability of data. The languages have a notion of *absence* meaning that there is no data available at a certain channel. If this is the case, no communication and no synchronization is necessary or even possible. The languages have the ability to directly describe independent execution in the synchronous model. This changes for imperative synchronous languages, because for them a reaction is defined by special statements in the control flow and synchronization of parallel execution is performed based on the reactions. This leads to a drawback of this imperative language, because due to the control flow the threads synchronize even if they do not communicate, since the synchronization is implicitly forced by the control-flow.

The contribution of this work is the extension of imperative synchronous languages by *clock refinement* to add more expressiveness to avoid over-synchronization. This kind of clock

refinement is a new idea in the context of synchronous languages. Nevertheless, it can be compared to other concepts which are present in some data-flow synchronous languages. The extension changes the language in that a reaction, i. e. a step of a QUARTZ program, can be split into (smaller) substeps. Since parallel threads do not necessarily see the substeps of another thread, they do not need to synchronize with the substeps and the threads can run independently until the whole reaction is finished. The possibility to refine a step into substeps is introduced with the notion of clocks that can be hierarchically declared in form of a tree. Thereby, each of the clocks define a new abstraction level and the model behaves on this level like the original language.

The clock refinement extension changes the execution model of programs. On the one hand, design decisions for extending the semantics of the original statements to clock refinement have to be discussed and made. These are e. g. the synchronization of parallel threads, the execution of delayed assignments, which originally take place in the *following* step, and the positions where abortion and suspension take place. On the other hand, this implies some consequences together with the synchronous abstraction and other properties of the language. These consequences are e. g. related to causality, where in the original language the causality is limited to a single step (what happens *before*), it is now extended to substeps and can also imply restrictions on the execution order of parallel threads. These consequences are discussed and with practical application in mind, it turns out that the full extension may be too difficult to be handled by programmers and compilers. Therefore, the extension is limited at some points to keep the features in a convenient shape for the programmers while reducing the complexity for compilers.

The semantics of QUARTZ is defined by structural operational semantics (SOS) rules. The semantics of the clock refinement extension is defined in the same way. However, since QUARTZ is only aware of one homogeneous sequence of reactions, each application of the rules defines one reaction. This is changed since the semantics now has to deal with steps related to different clocks and also the variable values change differently according to the clocks. Another challenge is to define the semantics in a way that it covers the independent execution of unrelated substeps.

A key feature of the design flow, which is implemented for QUARTZ, is the intermediate format which abstracts from the complex control-flow statements in the source language but still covers the whole system behavior. Since the clock refinement extension changes the execution model, the original intermediate format is not able to handle the extension. For this reason, the intermediate format needs to be extended first which is done by introducing clocks and changing the execution model. Finally, the compile algorithm needs to be redefined to translate programs correctly. In contrast to the original algorithm, which must only handle one kind of steps, the new one has to handle substeps and e. g. needs to place abortion and suspension conditions correctly. Another challenge is to define forks and joins for parallel threads correctly according to the defined semantics. Similar to the original compile algorithm for QUARTZ, the new one already solves the schizophrenia problems.

Synthesis denotes the final code generation phase where the system is translated to e. g. a hardware description language or a sequential programming language. This final phase is also described and evaluated for clock refinement. The main challenge there is to determine a good *schedule* of the substeps. For example, for hardware generation the clocks are mapped to trigger inputs which can be used by a scheduler to trigger the one or the other thread. As

already remarked, the clocks are not arbitrarily fed in from the outside and therefore, the scheduler has to respect the original control flow which defines the possible combinations of clocks. However, if parallel threads use the same resource (e. g. an multiplier), depending on the schedule, the multiplications can either be sequentialized to share a functional unit, or parallelized to use different functional units. Hence, in hardware either more *space* or more *time* can be used. For generation of software from the clock refinement extension, also the parallel dependencies have to be respected. This is easier for software because no trigger is needed for the execution of a thread: It is simply executed until data or synchronization is needed.

Finally, the introduced concept of clock refinement is evaluated by some case studies. Thereby, also features and advantages for programmers and the new obtained design freedom are discussed. For the clock refinement extension, the terms *extension* or *extended* QUARTZ are also used.

1.3 Outline

This thesis is structured as follows. Chapter 2 describes the related work for this thesis by introducing some of the models of parallel computation for embedded systems first, and by giving an overview of the synchronous language QUARTZ that is the foundation of the extension which is presented later afterwards. Thereby, the design flow for QUARTZ starting with the actual language description considering the semantic issues which come from the synchronous abstraction, and also compilation and code generation are discussed. In Chapter 3, some drawbacks of the language QUARTZ are first discussed, before then the idea of the extension is introduced by examples. Then, the impacts of the extension to the behavior of QUARTZ are addressed and discussed. The design flow for the language QUARTZ is then adapted to the extension, and therefore, Chapter 4 extends the definition of the original semantics to the extension, and Chapter 5 introduces an intermediate format and a compilation algorithm to translate programs of the extension to this format. Chapter 6 explains how the intermediate format can be translated to a hardware circuit. The extension and its use in the design of embedded systems is evaluated in Chapter 7 by considering an example. Furthermore, the chapter provides a discussion about the differences between the extension and the related work. The thesis concludes with a short summary in Chapter 8.

Chapter 2

Related Work

The content of this chapter is twofold. First, it introduces models of parallel computation for embedded systems as far as they are of interest for this work. Second, the imperative synchronous language QUARTZ is introduced in more detail since it is the foundation this thesis.

2.1 Models of Parallel Computation

Since traditional sequential programming languages are not directly suitable for the design of embedded systems, because the languages do neither offer constructs for periodic interaction, nor well-defined parallel computation. The better choice is to support these features directly by the computational model and with dedicated constructs in the languages [Berr89]. Many languages have been developed in the past which are mostly based on some computational models that basically determine when and how computations are executed and how data is communicated between concurrent parts [LeSa98, GiLL99, Jant04, ShTa10]. Some of the languages are introduced in this section.

2.1.1 Synchronous Model

The fundamental idea of synchronous systems [Berr89, BeBe91, BCEH03, PoST05] is to abstract from communication and computation delays and to consider *ideal* reactive systems producing their outputs instantaneously when their inputs arrive. This gives the programmer an ideal model where he or she can assume instantaneous interaction of components and can focus on the actual system's logic. Furthermore, the systems can be very well composed, described and analyzed. A reaction does not consume time in this abstraction, but from one reaction to the next one, *logical* unit of time is consumed. However, the simplicity of the programming model provided for the programmer results in more effort for compilers which now have to deal with the problems the programmer does not have to take care about [Nebu03, TaSi04, ScBS04b, JoPS10].

Obviously, this is an idealized view to programming and real computations take physical time, but the model is only used to describe the behavior of systems. It is then the task of compilers and tools to generate code that computes the described behavior, and it is up to the programmer or to further tools [LLBH05, MeHT09] to ensure that the real-time

constraints are met, hence that the computation is *fast enough* for the designated application. Since the whole reaction is considered as a single point of time, a signal can only have one value in each reaction. Even if this sounds like a restriction at the first glance, it is the key property leading to deterministic concurrency. Reactions are also referred to as *instants*.

The synchronous model is also present in synchronous hardware circuits that are therefore a good example for this model. Circuits are driven by a hardware clock of a fixed physical frequency triggering the registers of the circuit. In each clock cycle, input values can change and the values of local signals change accordingly. Obviously, there is a propagation delay of the electrical signal for gates and wires, and therefore, the clock cycle duration must be *long enough* to ensure that the values of the wires stabilize to their final value. The actual view to the system is simplified by just considering the value of a wire as it is defined by the functional behavior of gates, and the implementation (clock cycle duration) has to ensure that the value is correctly computed. However, not every circuit stabilizes to a final value, and therefore more effort for tools is needed to detect or remove such problems [Mali94, ShBT96, Edwa03a].

Some examples of synchronous languages are introduced in the following. Thereby, some of them do not require each signal to carry a value in each instant, and the signal is called to be *absent* in this case. Otherwise, to emphasize that a signal has a value, it is called *present*. A synchronous system (or language) which does not allow absent values is called a *strict synchronous system*, or otherwise it is called *non-strict*. Since the language QUARTZ, which is introduced later and which is the basis for the presented extension, is a strict synchronous language, the distinction is only used in this chapter.

Synchronous Data-Flow Languages

Note that this section is *not* about synchronous data-flow (SDF) as it is introduced by Lee and Messerschmitt in [LeMe87, LeMe87a]. The authors define in these works synchronous data-flow with process nodes consuming and producing a fixed number of tokens each time they fire. This behavior is better covered by the term *static data-flow*. The synchronous languages which are introduced in this work follow the synchronous assumption which means that the response of an operation instantaneously follows the occurrence of the inputs.

Synchronous data-flow languages implement the system's behavior by equations defining the values of outputs and local variables based on inputs and local variables. Since the execution of synchronous systems is an infinite sequence of instants, the equations define the value of a variable in each instant. In this section, the language LUSTRE [CHPP87, HCRP91, Halb93] is used as an example. As other data-flow languages, LUSTRE is also a non-strict synchronous language, i. e. it has a notion of presence and absence of values. However, it is restricted in a way that only the presence of data can trigger computations. It consists of data operators for manipulating values in and instant and of sequence operators for manipulating the presence, absence and values across instants. For example, the simple data equation

$$a = b + c;$$

defines the value of a as the sum of the values of b and c for each reaction. In addition to simple functional operators, which also force the clocks of the variables to be the same, other operators can also manipulate the clocks. Therefore, the variables can be considered as *sequences* of values where each value belongs to one instant. *Sequence operators* can be used to manipulate the values of streams:

- **pre** (x)

The *delay operator* accesses the last value of a stream. It can be interpreted as a shift operation for the stream:

$$x = (x_0, x_1, \dots)$$

then

$$\text{pre}(x) = (\text{nil}, x_0, x_1, \dots)$$

where `nil` means that there is a value at this position of the stream, but it is *not defined*. Note that this is different to an absent value meaning that there is no value.

- **x -> y**

Since it is not practicable to work with the value `nil`, the *followed-by* operator can be used to *overwrite* the first instant of a stream. Thus, for two streams:

$$x = (x_0, x_1, \dots) \quad \text{and} \quad y = (y_0, y_1, \dots)$$

the followed-by operator is defined by

$$x \text{ -> } y = (x_0, y_1, y_2, \dots)$$

These operators manipulate the values itself, but it is additionally possible to manipulate the presence and absence of values in instants:

- **x when B**

The sampling operator **when** can be used to select values of a sequence. When the value of `B` is `true`, the result is the value of `x`. Otherwise, the result is not present. The *clock* of the result is `B`. (`T` represents `true`, `F` represents `false`, and absent values are omitted in the following trace)

	1	2	3	4	5
x	x_0	x_1	x_2	x_3	x_4
B	T	F	T	T	F
x when B	x_0		x_2	x_3	

`x when B` is present when `B` is present and `true`. In this case, it has the value of `x`.

- **current** (x)

The projection operator **current** raises the clock of an expression to the next upper clock. If `x` has the clock `B`, then the clock of **current** (x) is the clock of `B`. The data values from `x` are used if `x` is present, otherwise the last value of `x` is replicated to fill the missing values.

	1	2	3	4	5
x	x_0	x_1	x_2	x_3	x_4
B1	T	F	T	T	F
B2	T	F	F	T	T
y1 = x when B1	x_0		x_2	x_3	
current (y1)	x_0	x_0	x_0	x_3	x_3
y2 = y1 when (B2 when B1)	x_0			x_3	
current (y2)	x_0		x_0	x_3	

In the example, the clock of y_1 is defined by B_1 and the clock of y_2 by $(B_2 \text{ when } B_1)$. The expression `current(y1)` raises the clock to the clock of B_1 and fills the gaps with the last values. In addition, the expression `current(y2)` raises the clock to the clock of $(B_2 \text{ when } B_1)$. Therefore, only the instants where B_1 holds are filled. Thus, only the value x_0 is added to the third instant.

```

node COUNT(init, incr: int; reset: bool)
  returns (n: int);
let
  n = init ->
    if reset then init else pre(n) + incr;
tel;

```

Fig. 2.1. LUSTRE Example: COUNT

```

node STABLE
  (set: bool; delay: int)
  returns (level: bool);
var count: int;
let
  level = (count > 0);
  count =
    if set
      then delay
    else if false -> pre(level)
      then pre(count)-1
    else 0;
tel

node TIME_STABLE
  (set, second: bool; delay: int)
  returns (level: bool);
var ck: bool;
var s_level: bool;
let
  s_level =
    STABLE((set,delay) when ck)
  level = current(s_level);
  ck = true -> set or second;
tel

```

Fig. 2.2. LUSTRE Example: TIME_STABLE

The clock of a signal identifies the instants in which the signal is present. The consistency of the clocks is a matter of LUSTRE: only signals with the same clock can be combined by a functional expression. The behavior is illustrated for a better understanding by two examples. The first example is a LUSTRE node COUNT from [CHPP87] given in Figure 2.1. It takes the inputs `init` and `incr` of integer numbers and `reset` of Boolean type and it produces the integer output `n`. In the first instant and in each instant where `reset` holds, the output `n` has the value of `init`. In all other instants, the previous value of `n` is taken and incremented by the current value of `incr`.

The second example TIME_STABLE originally comes from [LUSv6]. The node STABLE takes the inputs `set` and `delay`. Whenever `set` holds, for the number of instances given by `delay`, the output `level` is set. The node TIME_STABLE extends the behavior by adding

an additional input `second` to set the output `level` to `true` for the given number of *seconds* (number of instances where the inputs `second` is `true`). For this purpose, it can use the previously defined node `STABLE` and trigger it every instant where `second` holds or a new delay is given by `set`. The `current` operator is used to keep the value of `level` in between.

Polychronous Data-Flow

Polychronous data-flow languages are another way to specify synchronous systems in a data-flow manner. The formalism of polychronous data-flow is explained using the language `SIGNAL` [GaGB87, GGBM91, GuTL03, Gama10], but there exists also a graphical formalism `MRICDF` [JPSS09, JoSh10], and it can be also embedded in strict synchronous languages [BGSS13]. Like `LUSTRE`, it is also a non-strict language, and even if both look pretty similar, their behavior is different. For a `LUSTRE` node, any computation must be determined by the presence of data, but for `SIGNAL` also the absence of data can define a behavior. In this way, `SIGNAL` generally *specifies* multiple synchronous behaviors, but for real deterministic implementations, the system should only have one possible behavior. Furthermore, because `SIGNAL` is in a more declarative style, it challenges compilers [AmBG94, Nebu03, JoPS10] to generate executable code for a specification.

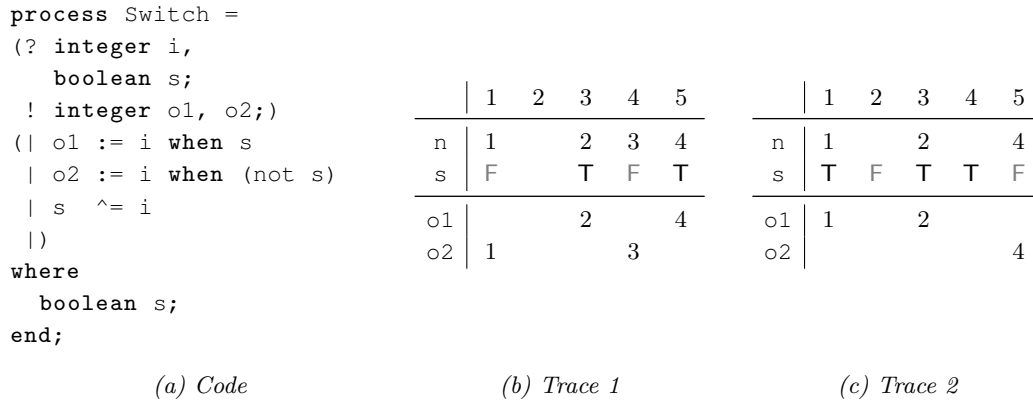
The behavior of a system is specified in `SIGNAL` by equations, and each operator imposes additional constraints on the clock of a variable, i. e. constraints on the presence and absence of values. Furthermore, clock constraints can be additionally given by the programmer directly. A simple data equation

$$a = b + c;$$

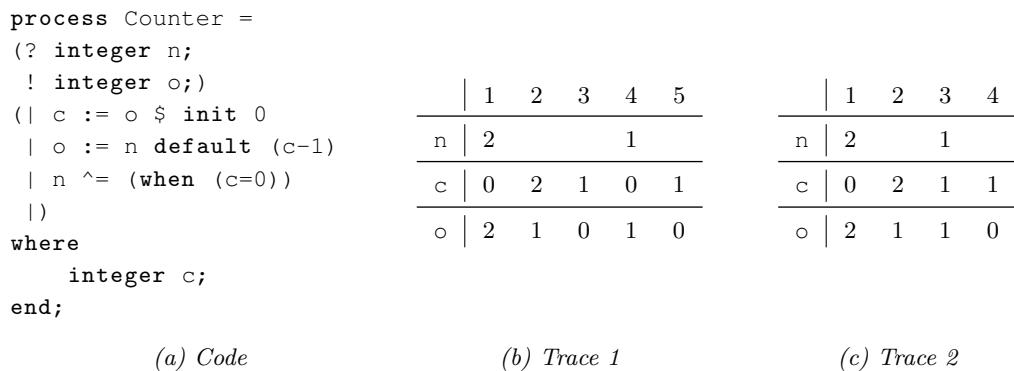
defines `a` as the sum of `b` and `c` in each instants when all three variables are present. It additionally imposes the clock constraint, that all three variables must have the same clock $a \hat{=} b \hat{=} c$. In addition to simple data operators, `SIGNAL` also has clock operators:

- `y := x $ init d`
The delay operator has one input `x` and produces the output `y` and stores the last value of `x`. Whenever `x` is present, `y` is also present and has the last value of `x`, except for the first time, then the value of `d` is used for initialization. The clock constraint imposed by the delay operator is $x \hat{=} y$.
- `y := x when z`
The downsample operator `when` selects the values of `x` when `x` and `z` are present and `z` is true, otherwise `y` is also absent. Note, that this operator does not require that `x` and `z` to have the same clock. The clock constraint $y = \text{when } \hat{x} \text{ and } \hat{z}$ and `z` is imposed by the `when` operator.
- `y := x default z`
The `default` operator performs an up-sampling. It takes the value of `x`, if `x` is present, otherwise it takes the value of `z`. The operator can also be seen as a merge with priority on `x`. The clock constraint imposed by `default` is $y \hat{=} x \hat{+} z$ which is the *union* of both clocks.

The idea of `SIGNAL` is illustrated by two examples. The first example is given in Figure 2.3 (a). The process `Switch` takes two inputs `i` and `s` and produces the outputs `o1` and `o2`. The

**Fig. 2.3.** SIGNAL Example: Switch

behavior of the process is that the values of the input i are forwarded to either $o1$ or $o2$ depending on s . The clock constraint $s \hat{=} i$ ensures that the inputs are present at the same time. The first example trace which is shown in Figure 2.3 (b) illustrates one possible behavior. Whenever s has the value true, the value of i is forwarded to $o1$. If it has the value false, the value of i is forwarded to $o2$. The second trace which is shown in Figure 2.3 (c) is not a valid behavior of the process, because the clock constraint is not fulfilled in the second and the fourth instant. If the clock constraint would be omitted, also this input combination would be allowed.

**Fig. 2.4.** SIGNAL Example: Counter

The second example is given in Figure 2.4 (a). It implements a simple counter which has one input n and one output o . The intention of the process is that for each present input value of n , the output values $n, n - 1, \dots, 0$ are produced. To this end, the local signal c stores the last value of the produced output, whereas o is produced by subtraction of 1 from c . However, when a new value for the input n is present, the output is updated by this value. The clock constraint $n \hat{=} (\text{when } (c = 0))$ ensures that a new input is only allowed to

be present when the local signal c reaches 0. The first example trace which is shown in Figure 2.4 (b) is a valid one and shows the desired behavior. First, 2 is present as input and the output produces the values 2, 1, 0. After that, the local signal c is 0 and a new input is present. The second trace which is shown in Figure 2.4 (c) is *not* valid, because the clock constraint is not fulfilled in the third instant. In this instant, the input value of n is not allowed to be present because c is not 0. Note, that without the given clock constraint both traces would be valid but the constraint selects just the first one to be valid. The SIGNAL example Counter also shows that the local signal c and the o can be present even if no input is present. This is called *over-sampling* [GuTL03] because the local signal or output are present more often than the inputs. A result for the implementation of this process is that it cannot be triggered by the inputs, because it also has a behavior when the inputs are absent.

ESTEREL

The imperative synchronous language ESTEREL [BeCo85, Berr00] is similar to the language QUARTZ that is introduced later in this chapter, and therefore, only a difference is pointed out here. It was said above that due to the synchronous abstraction, each signal can only have one value per reaction. This is also true for ESTEREL, but since it is an imperative language, it has some basic notion of sequential execution in a thread with well-defined bounds for the reaction. *Variables* can be used in addition to *signals* and they can have multiple values in one reaction, but with the restriction that variables cannot be shared to be read and written in different threads. Consider the following example:

```
X := 0;
emit S1(X);
X := X + 1;
emit S2(X);
```

The example is taken from [Berr00] and the variable X is assigned two times in one reaction and used to set the value of the signal $S1$ to 0 and the signal $S2$ to 1. Even though these ESTEREL variables are useful, they have the already mentioned limitation that they cannot be read and written in different threads, because in this case it would not be defined which read operation belongs to which write operation. Note that this issue is not present for signals, since they only have one value for each reaction, and therefore, it is well-defined which value is read and written. Furthermore, the variables cannot be used in loops to compute more complex functions.

An approach to loosen the restrictions was done in [HMAD13] where concurrent reads and writes for variables are allowed in different threads, as long as enough scheduling information can be obtained by the remaining program to match the reads and writes *and* to ensure that the write is executed before the read. It should be also mentioned here that this distinction of variables and signals is only made for ESTEREL. In QUARTZ, both terms are used synonymously for signals which can only have one value per step.

Statecharts and SyncCharts

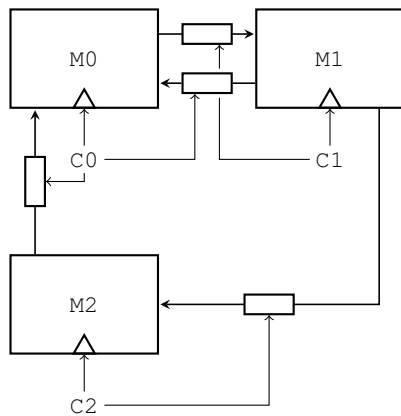
A further representative of the synchronous model are Statecharts that have been developed by Harel and Pnueli [HaPn85, Hare87] independently of the other languages. Statecharts are a graphical representation of control-flow with expressive constructs as they can be found in other imperative languages. Thereby, it comes with no surprise that Statecharts can be translated to ESTEREL and back [SSBD99, PrTH06], which is also the reason that Statecharts are not considered in more detail here. A second very similar formalism are SyncCharts [Andr95, Andr03].

Multiclock ESTEREL

Originally, ESTEREL also has the single-clock abstraction of steps, but in the past, it has been extended by two different multiclock extensions which are both named *multiclock* ESTEREL. They are introduced in [BeSe01] and [RaSh00a], and are discussed in the following.

Multiclock ESTEREL due to Berry and Sentovich

Berry and Sentovich introduced in [BeSe01] their idea of a multiclock extension for ESTEREL. While original ESTEREL programs are driven by a single clock, their work addresses the need to design systems with multiple clock domains in a modular way. Thereby, each module can run on its own clock and each step of the module coincides with a clock tick of the module's clock. The modules are still *single-clock* modules with the ability to call or abort other modules on a different clock. Additionally, two communication devices, named *sampler* and *recloner*, are selected by the authors to establish data communication across different clock domains. This extension is able to describe systems combined of modules each having another clock triggering its execution. The clocks have to be given from the environment, e. g. like a hardware clock.



(a) Plain (Module-)Structure

```

module M0 :
...
  abort
    run M1 clock C1 input ...
      output ...
  when ...
    run M2 clock C2 input ...
      output ...
...
end module

```

(b) Hierarchical (Code-)Structure

Fig. 2.5. Multiclock ESTEREL Structure

Figure 2.5 shows the structure of a Multiclock ESTEREL program. The right-hand side of the figure illustrates the code structure where the module `M0` calls `M1` and `M2`, each based on a different clock. Even though, the module `M0` starts and can also abort the submodules, the structure based on the clocks is plain, as it is illustrated on the left-hand side of the figure. The modules are triggered by their own clocks and communicate with the devices specified by the module call (additional control signals to start or abort the modules are also communicated). The example shows that even though the code is hierarchically organized with module calls, the clock structure is plain and that each module is just single clocked. Furthermore, the clock signals to trigger the execution have to be given from the environment.

The semantics of Multiclock ESTEREL is defined by modelling the extension in classical ESTEREL, where the global clocks are considered as additional inputs which have to be provided to the system. Thereby, the communication devices and also the module calls are translated. This translation allows to reuse all tools like simulators, available for ESTEREL.

Multiclock ESTEREL from Rajan and Shyamasundar

The second multiclock ESTEREL extension was published by Rajan and Shyamasundar in [RaSh00a, RaSh00, RaSh00b]. Their solution introduces a new statement `newtick` allowing to *override* the clock locally by an expression based on known signals. The local code block is then triggered by this new clock instead of the module's clock. The signals which are read inside such a clock domain are *latched*, thus, simply the last value of a signal is taken. Finally, the signals where the local clocks are defined with, have also to be provided by the outer environment. The difference to Berry's extension is that no dedicated clock signal is used, but any signal can be used to define a new tick. Without going into too much details and semantics, consider the following example [RaSh00a]:

```

module INVERTER :
  input I1
  output O1
  newtick I1 in
  loop
    await tick;
    emit O1(not ?I1)
  end
end
end

```

The new statement `newtick` overrides the basic clock with the input `I1`. Thus, the execution of the code block is now triggered by the signal `I1`. In ESTEREL, overriding the clock with a known signal in this way can only lead to a sampling of the already existing clock, since signals only change with the already defined clock itself. Even if the authors did not explicitly say this, it seems that they lose this property and allow a signal to change arbitrarily in a discrete-event manner. Otherwise, they would not be able to implement a Muller C-gate within their extension [RaSh00]. It rather seems that they are extending the model of discrete-event languages like VHDL with ESTEREL statements than extending the ESTEREL language.

Like Berry’s extension, this one also allows to define a new clock for a code block. However, it is not possible to access multiple clocks at the same time: only one clock is visible at each position in the code. Furthermore, the clocks are given from the outside without local restrictions to trigger computations.

2.1.2 Asynchronous (Untimed) Model

After the synchronous model was explained with some languages, this section now introduces asynchronous (untimed) models of computation. They do not rely on a global time trigger, but only describe the causality between actions.

CAOS

The first asynchronous model that is considered are Concurrent Action-Oriented Specifications (CAOS) [HoAr99, Arvi03, SiSh07, RSAS07] used to describe the data flow of hardware. These specifications describe how the state of the system is transformed by actions, but the decision when the actions are executed is left for the compiler and the execution. Since there could be several possibilities, the model is generally non-deterministic and a specification defines a set of possible behaviors. So the challenge for compilers is to translate this behavioral description to the Register Transfer Level (RTL) by scheduling independent actions into the same hardware clock cycle. Thereby, independent actions executed together do not lead to another behavior than executed in sequence.

SHIM

The Hardware/Software Integration Medium (SHIM) was proposed in [EdTa05, Edwa05b] to describe heterogeneous embedded systems. The language is thereby used to describe the hardware, the software and the communication established in a rendezvous-style with blocking read and blocking write similar to CSP [Hoar78, Hoar83]. A system in SHIM consists thereby of sequential processes using point-to-point communication through channels. The process reaching the read or write first, blocks until the counterpart of the communication reaches it also. Then, values are exchanged and the execution proceeds. Therefore, this model is more restrictive than the blocking read communication introduced by Kahn [Kahn74] for his process networks using non-blocking write and unbounded buffers for communication.

2.1.3 Discrete Event Model

The last model that is mentioned here is the discrete event model [CaLa08] where the execution of system parts is triggered by the occurrence of events, and which is traditionally implemented in hardware description languages like VERILOG [IEEE05a], VHDL [IEEE08], and SYSTEMC [IEEE05]. Events triggering the execution are e. g. signal value changes, or events of timers. The semantics of those languages is usually given by a simulation performing the following steps in each simulation cycle:

- When an event occurs, all processes that are *sensitive* to this event are triggered and their updates are *collected* but not yet executed.

- In the second step, all updates collected in the first step are executed synchronously. Some updates are visible directly, e. g. a signal change to a certain value, in the current point of time, other updates are scheduled for the future, e. g. a signal change after 5ns.
- If an update in the second step was done for the current point of time, the simulation is triggered again with this event. Technically, the simulation time does not increase in this case and the cycle is called a δ -cycle. If no new event occurred for the current simulation time, the time advances until the next event occurs.

The constructs used in those languages are useful for hardware simulation, but only a subset of the whole languages is synthesizable to real hardware. Furthermore, when hardware synthesis is the goal, the systems described in those languages are often synchronous systems, i. e. the only event that triggers the execution is a hardware clock signal.

2.2 The Synchronous Language QUARTZ

This section introduces the imperative synchronous language QUARTZ, which is used as a basis for the extension in this work. Thereby, the section is mostly based on [Schn09], which provides an exhaustive introduction to QUARTZ and serves as a reference for the whole section. Since most of the content is recapitulated in a later chapter in the context of the extension, this section gives an informal overview to introduce the language, its semantics and its compilation. Compiler, verification and synthesis tools for QUARTZ are implemented on the basis of the AVEREST library [AVEREST].

The imperative synchronous language QUARTZ implements the synchronous model of computation by means of the `pause` statement. While all other primitive statements do not take time (in terms of macro steps), a `pause` marks the end of a macro step and consumes one logical unit of time. Thus, the behavior of a whole macro step is defined by all actions between two consecutive `pause` statements. Parallel threads run in lock-step: their macro steps are executed synchronously, and the statements in both are scheduled according to the data dependencies so that all variables have a unique well-defined value in the macro step. QUARTZ is a strict synchronous language, thus, each variable has a value per step and there is no notion of absence.

2.2.1 Introductory Examples

The syntax and behavior of QUARTZ programs is illustrated by two examples in this section. The first example is the module `M` shown in Figure 2.6 (a). For the module, the both inputs `a`, `b`, the both outputs `x`, `y`, and the local variable `z` is defined. The `pause` statements are annotated with the *labels* 11, 12, and 13. An example execution trace is given in Figure 2.6 (b) for sample inputs: each column gives the values for one of the first 6 execution steps. Generally, the module is executed for infinitely many steps, and it gets a new input value for each input and it produces a new output value for each output in each macro step. For space reasons, the values `true` and `false` are abbreviated by `T` and `F` in the trace. The very first macro step is executed from the beginning of the module until the first `pause` statement is reached. Therefore, the variable `x` gets its value in the first step accordingly to the given input value of `a`. Furthermore, the variable `z` is assigned by a so-called *delayed assignment*,

```

module M(nat ?a, ?b, x, y)
{
  nat z;
  loop {
    x = a;
    next(z) = b;
    l1: pause;
    x = y + z;
    if(a > 4) {
      y = b;
      l2: pause;
    }
    l3: pause;
    z = 3;
  }
}

```

(a) M Source Code

		1	2	3	4	5	6
Inputs	a	2	5	1	1	2	2
	b	5	2	1	3	4	1
Labels	st	T	F	F	F	F	F
	l1	F	T	F	F	T	F
	l2	F	F	T	F	F	F
	l3	F	F	F	T	F	T
Locals & Outputs	z	0	5	5	3	3	3
	x	2	7	7	1	5	2
	y	0	2	2	2	2	2

(b) Sample Execution Trace

Fig. 2.6. M Example

which evaluates an expression in the current step, but the value is transferred to the following step. If a variable is not assigned in a step, it gets its default value, as it is the case for the variable y , but also for the variable z , because the delayed assignment defines only the value for the following step. For the next step, the execution starts from the first `pause` statement, and because of the input value a , the `if` statement is entered and the step ends at the `pause` statement with label `l2`. According to the synchronous model, each variable has a unique value for the entire step and all assignments are evaluated simultaneously. For now, the assignment to y must be executed *before* the assignment to x , because its value is required to evaluate the expression of the assignment. The variable z gets the value that was defined by the delayed assignment in the step before. Since no assignment is executed in the third step, all values of the variables are kept from the previous step. In addition to the variables, the labels of the `pause` statements are also shown in the trace. The labels are interpreted in the following way: when a label is set to `true`, the current step starts at the associated `pause` statement. Since there is no label that holds for the first step (the program has just been started), the implicit label `st` is introduced to indicate the start of the module. This label only holds in the first step.

A second, more complex example, is the module `ABRO`, which was originally published as an `ESTEREL` example by Berry [Berr97a], and whose `QUARTZ` version is shown in Figure 2.7 (a). The implementation uses some more complex statements, whose behaviors are explained now as far as needed for this example. The `await` statement defines, similar to the `pause` statement, the end of a macro step. However, in contrast to the `pause` statement, the `await` statement is only left in one of the following steps, when the given condition holds. Actually, it is a *macro statement* of other primitive statements defining its behavior:

$$l: \text{await}(\alpha) \quad \equiv \quad \text{do} \{ l: \text{pause}; \} \text{while}(!\alpha).$$

```

module ABRO(event ?a, ?b, ?r, !o)
{
  loop {
    abort {
      wa: await(a);
      ||
      wb: await(b);

      o = true;
      wr: await(r);
    } when(r);
  }
}

```

(a) ABRO Source Code

	1	2	3	4	5	6	7	8
a	F	F	T	T	F	F	T	F
b	F	T	F	F	T	F	F	T
r	F	F	F	T	F	T	F	F
st	T	F	F	F	F	F	F	F
wa	F	T	T	F	T	T	T	F
wb	F	T	F	F	T	F	T	T
wr	F	F	F	T	F	F	F	F
o	F	F	T	F	F	F	F	T

(b) Sample Execution Trace

Fig. 2.7. ABRO Example

The step ends in any case, but in the following steps, the execution can only proceed after the statement when the condition α holds, otherwise, the loop is restarted and the step ends again. The `abort ... when` statement cancels the execution of the enclosed statement when the given condition holds. The ABRO module gets the three inputs `a`, `b`, `r`, and produces the output `o`. The behavior of the module is as follows: it waits for the inputs `a` and `b` in parallel (`||`) and only if both have occurred, the output `o` is set to `true`. This behavior can be reset by the input `r`, which is used for the `abort` statement to cancel the execution. In this case, the surrounding `loop` restarts immediately. The behavior can be described by the *extended finite-state machine (EFSM)* which is shown in Figure 2.8. The labels which occur in the program are used to define the states of the EFSM. From the initial state, the first transition goes to the state `{wa,wb}` where the system waits for `a` and `b` to become `true`. If the input `a` holds in the next step, the system goes to state `{wb}`, where it just waits for the occurrence of `b`. If then `b` holds, the system goes to state `{wr}` and the output `o` is set to `true`. Thus, `a` and `b` have occurred and then the output `o` is set. From each state, the occurrence of the signal `r` lead to a reset of the system and it will go back to state `{wa,wb}`. Note that, since the output `o` is declared as an `event`, it will only remain `true` for the step where it is explicitly set. A sample execution of the ABRO example is given in the trace in Figure 2.7 (b) for some sample input values.

The ABRO example also shows the correspondence of labels in the source code and the states of the EFSM. Hence, the labels encode the (control) states, but not each combination must be necessarily reachable. In the example, all combinations of the labels `wa` and `wb` occur in the EFSM. The example only waits for the two inputs `a` and `b` in parallel, but it could be extended to any other number of input values. In this case, for each additional input, a new label and an `await` statement are introduced and all combinations of those labels would encode a reachable state of the EFSM. This would blow-up the states of the EFSM whose number grows exponentially with the number of inputs. Thus, in the worst-case, the EFSM can grow exponentially in terms of size of the source code.

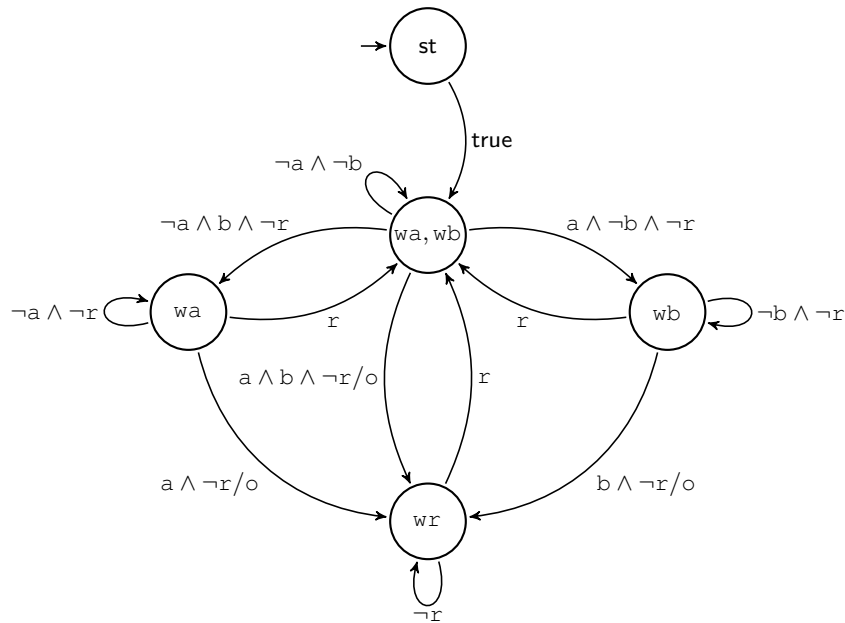


Fig. 2.8. EFSM of the ABRO Example

The module M also shows how variables can be declared either as interface variables or as local variables. If the variables are declared in the module's interface, they are implicitly declared as input-output variables. If they are annotated with an $?$, they are declared as inputs, and if they are annotated with an $!$, they are declared as outputs. Input-output variables are important for the interaction of several modules, but they can be seen as outputs (which can be read) for the top-level module. Therefore, both kinds, input-output variables and outputs, are simply referred to as outputs in the following. Additionally to this distinction, there are two kinds of variables: *events* and *memorized* variables. Thereby, the keyword `event` is added to define a variable as an event, otherwise the variable is declared as a memorized variable implicitly. Both kinds of variables differ in the behavior when they are not explicitly set by an assignment. If an event variable is not set by an immediate assignment in a macro step and neither by a delayed assignment of the previous macro step, the variable automatically gets its default value. A memorized variable will keep its value of the previous step (like a register).

2.2.2 Statements

The language QUARTZ basically consists of a set of *core statements* and a set of *macro statements*. Thereby, the semantics of the core statements is directly defined, and the macro statements are reduced to the core statements. The exhaustive list of statements can be found in [Schn09]. Here only the statements are considered which are important for this work. The statements which are not considered here are either used for verification or they can be also defined by macros from the core statements which are listed below. Some of the core statements are:

- **l : pause**
The **pause** statement marks the end of a macro step and thus also the beginning of the following step. An optional label *l*, which is used to identify the **pause** statement in the source code, is annotated to the statement. The label also plays a role for encoding the control flow, e. g. in an EFSM or for code generation. Therefore, the compiler adds an implicit label if none is given by the programmer.
- **x = τ**
The *immediate* assignment sets a value of a variable in the current macro step. Therefore, the expression τ is evaluated under the values of the current step, and the result is assigned to the variable *x*. All assignments which are executed in a particular step are evaluated synchronously.
- **next(x) = τ**
The *delayed* assignment sets the value of a variable in the next step. Like the immediate assignment, it evaluates the expression τ with the values of the current step, but the resulting value is assigned to the variable *x* in the following step.
- **if(σ) ... else ...**
The conditional statement executes its first or its second branch depending on the evaluation of the condition σ . The condition is only checked when the statement is reached by the control flow, it is not evaluated again when control flow already is inside one branch.
- **{ α x; ...}**
Local variables are declared at the beginning of a code block and they are visible for this code block. The block, where a variable is visible is called its *scope*. Input and output variables are visible for the whole module. The statement defines the variable *x* of type α for the code block. Local variables can be only read and written within their scope.
- **do ... while(σ)**
The loop repeats the execution of its body statement if the condition σ holds. Therefore, the body is executed at least once, when the end of the body is reached, the condition σ is checked and if it holds, the loop body is restarted in the same step. Otherwise, the whole loop terminates.
- **{ ... } || { ... }**
The parallel statement executes both code blocks, called *threads* for this statement, stepwise synchronously. Hence, in each macro step, one macro step of each block is executed. In plain terms, both threads synchronize on each **pause** statement which is reached. The whole parallel statement terminates when its last (in terms of execution) thread terminates.
- **abort ... when(σ)**
The abortion statement cancels the execution when the given condition σ holds. In this case, program execution will proceed after the statement. The abortion takes place for the whole macro step: if the condition holds in a step, no action inside of the code block of the abortion statement is executed. This version of the abortion is also called *strong* abortion when the difference to the *weak* abortion which is explained below is emphasized.

- **immediate abort ... when(σ)**

The abortion statement above can only abort the execution, when the control flow is already inside the statement. The **immediate** keyword changes the behavior so that also the first macro step, i. e. when the statement is entered, can be aborted. This statement can be also defined as a macro of the former one:

```
immediate abort ... when( $\sigma$ )  ::=  if(! $\sigma$ ) { abort ... when( $\sigma$ ); }
```

Hence, if the abortion takes place at the beginning, the whole statement is not executed, as it is illustrated by the macro definition.

- **weak abort ... when(σ)**

The *weak* version of the abortion statement also aborts the execution in the macro step in which the condition holds, but the assignments for the current step inside the code block are executed before execution proceeds after the statement. Hence, the data flow is executed, but the control flow is aborted inside.

- **weak immediate abort ... when(σ)**

Like for the strong variant, the **immediate** keyword changes the behavior that also the first macro step, i. e. when the statement is entered, can be aborted. However, for this weak version, the data-flow assignments of the entering macro step are executed.

- **suspend ... when(σ)**

The suspension statement stops the execution of the code block when the given condition σ holds. In this case, nothing will be executed for this step, but the control flow will rest at the label where it is. Thus, in the following step the execution resumes from exactly the same position (given that the condition σ does not hold and the execution is not suspended again). Like for abortion, this version of the statement is also called *strong* suspension in contrast to the *weak* versions.

- **l: immediate suspend ... when(σ)**

The **immediate** version of the suspension statement *waits* before entering for the condition α to not hold. Hence, the entering of the statement can be also suspended. In this case, an additional label is needed, because the control flow can rest outside the actual statement. The macro definition illustrates this behavior:

```
l: immediate suspend ... when( $\sigma$ )  ::=  while(! $\alpha$ ) { l: pause; }
                                     { suspend ... when( $\sigma$ ); }
```

- **weak suspend ... when(σ)**

The *weak* version of the suspension statement stops the execution of the control flow but not of the data flow when the condition σ holds. Hence, assignments are executed inside the code block, but the control flow remains at the labels it is. In a following step, when σ does not hold, it will resume exactly from this position, where it has been suspended before.

- **l: weak immediate suspend ... when(σ)**

Also for the weak version of the suspension statement, the **immediate** keyword changes the behavior in that suspension can also take place for the first macro step. However, the data-flow assignments of the entering macro step are here also executed.

- **nothing**

This statement has no effect. It exists for technical reasons of defining source code transformations and macro statements.

Note that the core statements, as also explained in [Schn09], do not form a *minimal set* of statements, because some of them can be defined by others, like the immediate versions of the strong abortion and suspension statement. This kind of statements, the abortion and suspension statements, are summarized by the term *preemption* statements. Furthermore, the preemption statements without the keyword **immediate** are referred to as *delayed preemption* if the difference is emphasized. So far, just core statements have been described. Furthermore, the following macro statements are of interest for this work:

- `loop { ... } ::= do ... while(true)`

This simple version of a loop keeps repeating its loop body forever. It can only be aborted by a surrounding abortion statement or stopped by a surrounding suspension statement.

- `while(σ) { ... } ::= if(σ) { do ... while(σ); }`

This loop also checks the loop condition before entering the loop body for the first time. It does not necessarily execute its body as the original `do ... while` loop.

- `l: await(α) ::= do { l: pause; } while(! α)`

The already mentioned **await** statement waits for the expression α to become true. In any case, the statement marks the end of a macro step and has therefore a label assigned to it. The execution only proceeds when the given condition holds. The macro definition shows this effect, the loop and the **pause** statement is reached in any case, but the loop is only terminated when α holds.

- `l: immediate await(α) ::= while(! α) { l: pause; }`

The **immediate** version of the **await** statements also checks the expression α in the first step, as it is illustrated by the macro.

As already said, this list of statements is not complete, but it gives an informal overview of the statements which exist in QUARTZ and which are important for this work. The statements will be considered later again in the context of the extension.

2.2.3 Semantic Issues

The synchronous assumption requires that all actions are executed simultaneously in each instant considering the same variable environment. This abstraction simplifies description and analysis on the one hand, but also introduces special problems being discussed in the following on the other hand. Thereby, *logical correctness* and *causality* are common aspects of synchronous languages in general, whereas *schizophrenia* problems are an issue of imperative synchronous languages in particular.

Constructivity vs. Logical Correctness

In QUARTZ, all actions executed in a step, i. e. between two **pause** statements, define the behavior in this step. In synchronous data-flow languages, the behavior is defined by a set of equations considered in each reaction. In theory, every variable assignment that complies

to the execution of the actions or to the equations can be considered as a valid reaction. A program that has for given input values exactly one consistent assignment of all variables is called *logically correct*. This is also exactly the form of determinism that is expected of reactive systems. However, the goal is to execute the program and therefore, the unique solution has to be computed somehow. Instead of finding the solution for each possible program, a subset of programs called *constructive* programs is defined for which the solution can be computed by a set of operational rules. The idea is illustrated with the help of the following QUARTZ program.

```

11: pause;
if (x | y) {
  x = true;
} else {
  x = true;
  12: pause;
}
y = true;
13: pause;

```

Consider the step that starts from label 11 and assume that the variables memorized x and y had the value `false` in the previous step. If no assignment sets them in the considered step they will keep their values of the previous step. In order to be logically correct, a unique variable assignment has to be found leading to a valid execution of the program. In each branch of the `if` statement, an assignment sets x to `true`, but the second branch also contains a `pause` statement. Hence, if the condition $x \mid y$ does not hold, the assignment to y is not executed, but the condition depends on the value of x . In principle, all possible variable assignments for x and y have to be checked.

It can be easily seen that only the values $x = \text{true}$ and $y = \text{true}$ are consistent with the executed assignments, and thus, this program (or at least the considered step) is logically correct. However, executing this program means to compute the values for x and y , and trying out all possibilities is obviously too inefficient. Therefore, the QUARTZ semantics is given by a set of rules defining an operational way of constructively computing the values. Thereby, a program where all values can be computed by these rules is called *constructive*, so that the above example is not a constructive QUARTZ program. In QUARTZ, an action is only executed if all control-flow conditions contributing to its trigger can be evaluated before that action. Since the condition that determines whether to execute the assignments to x and y depend on the values of x and y , they cannot be evaluated before.

An exact definition of constructivity is given by the semantics of QUARTZ, but for a better understanding, some examples of constructive programs are informally discussed in the following. Consider the examples shown in Figure 2.9. The program `Caus1` produces the output `o` and uses the local variable x . Since both assignments are executed in the same step under the same condition (they have the same control-flow condition), they can be executed in any order, but for the assignment to `o` the value of x needs to be computed before. Hence, the execution following the data dependencies can compute all values and the program is constructive. The program `Caus2` also produces the output `o`, but the execution of the assignment to `o` depends on its own value: the control-flow condition cannot be determined

<pre> module Caus1(bool o) { bool x; o = x; x = true; } </pre> <p style="text-align: center;"><i>(a) Caus1 Source Code</i></p>	<pre> module Caus2(bool o) { if (!o) pause; o = true; } </pre> <p style="text-align: center;"><i>(b) Caus2 Source Code</i></p>
--	---

Fig. 2.9. Causality Examples I

without knowing the value of o . One has to try out all possible values, but this is not considered to be constructive. Even more, one would find out that both possibilities (true and false for o) are consistent values for this program: the program is not even logically correct.

The program Caus1 showed a dependency between the assignments that can be statically resolved. However, consider the programs Caus3 and Caus4 in Figure 2.10 which have both cyclic dependencies between both assignments. Thereby, Caus3 is logically correct, since for each possible input value for i , there exists one unique consistent valuation for x and y . It is furthermore constructive in the sense of QUARTZ, since with a known input value, the cyclic dependencies disappear after a lazy evaluation based on i . The program Caus4 is similar, it also takes the input i and produces the same outputs, but it is not logically correct since there is no solution for input $i = \text{false}$. Lazy evaluation with that input assignment will also not resolve the dependency cycle between x and y . Finally, a third kind of causal dependencies is illustrated by the program Caus5 that is also shown in Figure 2.10. The first thread consists of a loop needing one macro step and the second one consists of a loop needing two macro steps for executing their body statement. Due to the synchronization of the parallel statement, the assignments of the first thread are executed alternately with the one or the other assignment of the second thread: the dependencies between the assignments in the first thread are changed by the second thread, since the order depends on whether the one or the other assignment is executed. However, the dependencies can be statically determined in each macro step, but depend generally on the control-flow location. The program is also constructive in the sense of QUARTZ.

The demand of constructivity comes from the fact that synchronous languages are used to describe systems being finally executed, but real execution needs an operational description of the system. Even though, compilers could transform each logically correct system into a constructive form, the question is whether it is a good idea to spend this computational effort. Furthermore, Shiple and Berry [ShBT96, Berr99] defined a constructive semantics for pure ESTEREL and showed that the hardware circuit being generated (by a simple translation) for a program stabilizes exactly when the program is constructive. Hence, on the one hand, the definition of constructivity can be seen as a restriction that allows one to more easily translate programs to a target platform. On the other hand, it also gives the programmer an understanding about how things are computed. Checking constructivity statically is known as *causality analysis* [Mali94, HaMa95, BrSe95, ShBT96, Bous98, Berr99, ScBS04b, SBST05b] in the context of synchronous languages. Furthermore, even if some kinds of dynamic

```

module Caus3(bool ?i, x, y)
{
  x = ! y & i;
  y =  x | i;
}

```

(a) Caus3 Source Code

```

module Caus4(bool ?i, x, y)
{
  x = ! y | i;
  y =  x | i;
}

```

(b) Caus4 Source Code

```

module Caus5(...)
{
  loop {
    x = a;
    y = b;
    pause;
  } || loop {
    a = y;
    pause;
    b = x;
    pause;
  }
}

```

(c) Caus5 Source Code

Fig. 2.10. Causality Examples II

dependencies like in the program `Caus3` can be also resolved by hardware circuits, a translation to a sequential language requires more effort in this case.

Local Declarations and Schizophrenia

Imperative synchronous languages combine control-flow statements with synchronous semantics for data flow with the result that assignments can affect a finite number of statements in the source code. Local declarations restrict the visibility of a variable to a certain scope. Together with loops, it is possible that the scope of local variables is left and re-entered in the same macro step [Berr99].

The example program `Local1` shown in Figure 2.11 (a) illustrates this aspect. It contains a loop with a single `pause` statement of label `1`, and a declaration of the local variable `x`. Since the scope of `x` ranges from its declaration to the end of the loop body, `x` exists for each execution of the body. However, a step can start from the `pause` statement, complete the loop, re-enter the loop, and finally end at the same `pause` statement: the scope of `x` is left and entered in one macro step. Thereby, the assignment `x = (y < 1)` affects the variable in the *old* scope, whereas the `if` statement reads the value of the *new* scope, which is `false` because `x` is initialized each time the scope is entered and `x` is not written. The variable `x` exists twice in this macro step.

A second example is shown in Figure 2.11 (b). The program `Local2` also contains a single loop with the local variable `x`. A macro step inside the loop can either start from label `11` or `12`. If it starts from `11` and the input `i` holds, it will also end at `11`, and the assignment `x = true` is executed for the `x` whose scope is left in this step. If a macro step starts from `12` and the inputs `i` does not hold, the step will end at `12`, and in this case, the assignment `x = true` is executed for the *new* `x` whose scope is entered in this step. Thus, the statement `x = true` depends dynamically on either the one or the other version of `x`. Such statements are called *schizophrenic statements*.

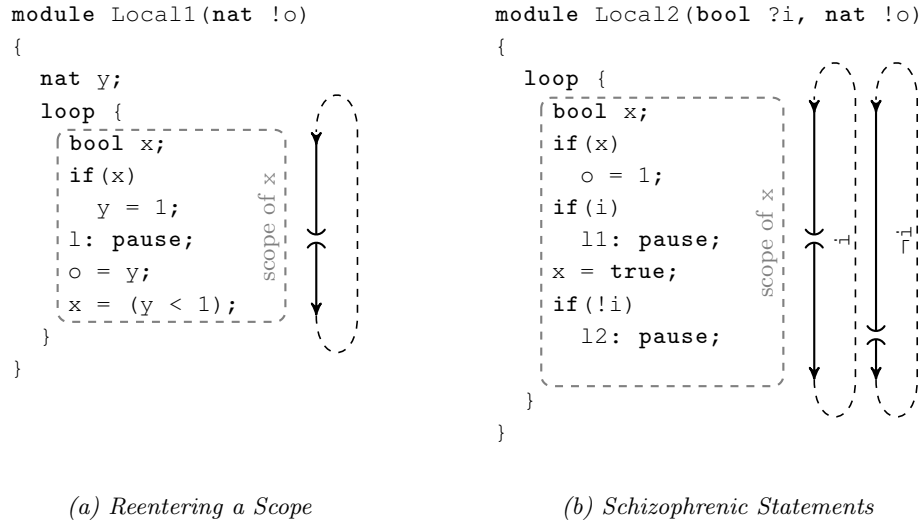


Fig. 2.11. Schizophrenic QUARTZ Programs

Compared to local declarations in ordinary sequential programming languages, it seems curious that those examples are an issue in synchronous languages, because when the scope of a variable in a sequential language is left, the variable cannot be used anymore. If a scope is entered, the variable can be just reset, or dependent on the implementation new memory is reserved for this variable. However, due to the synchronous abstraction, values of local variables can affect the execution of the whole step, hence, their value must be available. Consider the example in Figure 2.11 (a) again, where the value of the *old* x depends on y , but y depends on the *new* x . Finally, synthesis and analysis tools have to deal with local variables.

The shown examples only illustrate simple occurrences of schizophrenic variables and statements where only two versions of a variable are needed for a macro step. However, nested loops in combination with the `weak abort` statement can lead to arbitrary complex constructs where also arbitrary (but finitely) many versions of the local variable are needed for a macro step. Schizophrenia is properly handled by the QUARTZ compiler, and also other solutions have been proposed in the past to handle local declarations for ESTEREL. One simple solution is a source-code transformation, which changes the code in a way that it is no longer possible to leave and enter the scope of the same variable in the same macro step. For example, the loop body is duplicated and together with it also the local declarations. This results in more local declarations and increases the code but it prevents the usage of the same variable twice in a step. However, for nested loops, this solution leads to an exponential growth of the source code. A better solution is proposed in [TaSi04], where the local variables and just *some* of the statements in the loop body are duplicated, but not the whole loop. However, the duplication approach does not work directly for QUARTZ, because QUARTZ has delayed assignments, which can affect the value of a variable in the following step. The duplication approach just prevents the body to be re-entered directly but not to be entered in the next step where delayed assignments could affect a variable. Therefore, a

third copy would be necessary for QUARTZ to handle this correctly. A third solution, which was proposed in [YKSH09] should be mentioned. The authors deal with a graph reachability approach to identify local declarations which are not a problem for *their* further processing. This solution does not tackle the problem generally, but only forbids examples like the one shown above with existing dependencies from the new to the old variable. Further discussions and solutions can be found in [ScBS06].

2.2.4 Formal Semantics

The semantics of QUARTZ is formally defined in [Schn09] in the style of Plotkin’s *Structural Operational Semantics (SOS)* [Plot81, Moss06] which has already been successfully used in the context of the synchronous language ESTEREL [BeCo85, Berr99, Tini00], and has been also utilized for the polychronous language SIGNAL [TBGS13]. As the name suggests, SOS rules are defined over the structure of a given program, i. e. the *Abstract Syntax Tree (AST)*.

In sequential programming languages, a program is executed step by step as given in the source code. The traditional SOS rules update a *state* according to the execution of a statement directly. For example, an assignment changes the value of a variable to the value given by the evaluation of the assigned expression with respect to the current state. The semantics of a sequence is given by the *sequential execution* of those statements and the behavior of a program can be determined by traversing the AST according to the rules once, because an assignment can only affect statements which occur after itself in the AST. However, due to the synchronous abstraction of time, all assignments of a macro step have to be taken into account at once. Hence, the SOS rules cannot be used directly, because the order of the statements given by the AST does not follow the semantic order. Therefore, the semantics of QUARTZ is described by two sets of SOS rules: *transition rules* and *reaction rules*. The reaction rules determine the value for each variable of a macro step, and the transition rules perform a transition of the program for the macro step.

Transition Rules

The transition rules transform a given program based on a valid variable assignment for the current step to a new program which is considered in the following macro step. The values are *stored* in an *environment* \mathcal{E} assigning a value to each variable. Transition rules for QUARTZ are of the form:

$$\langle \mathcal{E}, \bar{h}, \mathcal{S} \rangle \rightarrow_{\mathcal{Q}} \langle \bar{h}', \mathcal{S}', \mathcal{A}^{\text{next}}, t \rangle$$

The rules start with the given environment \mathcal{E} and an incarnation level \bar{h} which counts the already processed local declarations for a variable: every time a new scope of a local variable is entered, it is increased for this variable. With this information, expressions can be evaluated according to the current scope of local variables. Note that a scope can be entered more than once during a macro step. The statement \mathcal{S} denotes the program statement the current macro step is executed for. The rules determine the residual statement \mathcal{S}' , which is considered in the following macro step and a set of delayed assignments $\mathcal{A}^{\text{next}}$, which determine values for variables in the following macro step. The symbol t is either *true* or *false* and denotes if the current macro step is completed for the considered statement. Frankly, it is *true* when a *pause* statement has been reached and the whole macro step is processed. The environment

which is given to the transition rules has to be complete, i. e. each variable has a valid value. For the following macro steps, only the residual statement \mathcal{S}' and the delayed actions $\mathcal{A}^{\text{next}}$ are of interest. h' and t are only used during the transformation for traversing the AST.

Reaction Rules

In contrast to the transition rules modifying the program itself, the reaction rules determine the values of the variables of a macro step, thus, they prepare the application of the transition rules. However, due to the already mentioned reasons, the data values cannot be determined by traversing the AST once. Therefore, the reaction rules start with a partial environment, i. e. not all variables have a value assigned, and they are repeatedly applied until either all values for the variables are known or no more values can be derived. For this reason, the reaction rules cannot modify the program structure, because otherwise a repetitive rule application for the same step would not be possible. Reaction rules for QUARTZ are of the form:

$$\langle \mathcal{E}, h, \mathcal{S} \rangle \mapsto_{\mathcal{Q}} \langle h', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, t_{\text{can}}, t_{\text{must}} \rangle$$

The rules start with the given (partial) environment \mathcal{E} and an incarnation level h . The rules consider the statement \mathcal{S} , and they determine a new incarnation level h' , two sets of actions \mathcal{A}^{can} and $\mathcal{A}^{\text{must}}$ and two versions of a Boolean flag t_{can} and t_{must} . The actions $\mathcal{A}^{\text{must}}$ are the actions which are known to be executed in the considered step, whereas the actions in \mathcal{A}^{can} are potentially executed. An action is for example contained in \mathcal{A}^{can} when the action occurs within an `if` statement, and its condition cannot yet be evaluated to `true` or `false` because the needed variables are not known. According to the assignments which are collected in those sets, the environment can be updated for the following iteration. The flags t_{can} and t_{must} are used during the application: they encode the information if a statement will or will not terminate in the macro step. According to the partial environment, it can be also the case that it is not known that the statement will terminate, which is also encoded in those flags. However, h' , t_{can} and t_{must} are only used during the application, and \mathcal{A}^{can} and $\mathcal{A}^{\text{must}}$ can be used to update the environment for the next iteration.

The reaction rules also define the set of constructive QUARTZ programs: a program where the values of all variables can be computed by these rules is said to be constructive. If the reaction rules cannot determine a value for each variable, the program can still be logically correct, but it is not causally correct.

Example

A definition of an interpreter for QUARTZ programs based on the SOS rules is given in [Schn09]. Figure 2.12 shows an example program that is interpreted using the rules in the following. The considered statement is \mathcal{S} and the program has the input `i` which can be `true` or `false`. The value `true` is considered first and the reaction rules are applied to \mathcal{S} :

$$\langle \{(i, \text{true}), (x, \perp), (y, \perp)\}, h, \mathcal{S} \rangle \mapsto_{\mathcal{Q}} \langle h', \{x = y, y = \text{true}\}, \{x = y, y = \text{true}\}, \text{false}, \text{false} \rangle$$

The symbol \perp is used to represent unknown values. The result of the rule means that the assignments to `x` and `y` are executed, but the value of `x` cannot be yet determined because the value of `y` is unknown. The environment is updated according to the assignments and the rules are applied again:

```

module S(bool ?i, x, y)
{
  x = y;
  if(i) {
    y = true;
    11: pause;
    x = false;
    12: pause;
  }
  13: pause;
  y = false;
}

```

```

x = false;
12: pause;
13: pause;
y = false;

```

Fig. 2.12. Example for QUARTZ SOS rules

$$\langle \{(i, \text{true}), (x, \perp), (y, \text{true})\}, \bar{h}, \mathcal{S} \rangle \rightarrow_{\mathcal{Q}} \langle \bar{h}', \{x = y, y = \text{true}\}, \{x = y, y = \text{true}\}, \text{false}, \text{false} \rangle$$

There is no change in the result of the rules, but since the value of y is now known, the value of x can be determined. Another application of the rules leads to the same result and the final environment is: $\{(i, \text{true}), (x, \text{true}), (y, \text{true})\}$. This environment can then be used to transform the program statement with the transition rules:

$$\langle \{(i, \text{true}), (x, \text{true}), (y, \text{true})\}, \bar{h}, \mathcal{S} \rangle \rightarrow_{\mathcal{Q}} \langle \bar{h}', \mathcal{S}', \{\}, \text{false} \rangle$$

Hence, the residual statement \mathcal{S}' is the one considered for the next reaction and can be also found in Figure 2.12.

2.2.5 Intermediate Representation

QUARTZ programs are translated by the compiler to the AVEREST *Intermediate Format* (AIF) [BrSc09, BrSc11a]. AIF is also based on the synchronous model of computation and contains the entire behavior of the given synchronous program. An AIF file contains some structural information about the source program such as the input/output interface, local variable declarations and verification targets. This information is not presented in this work and the focus is put to the functional description. The intermediate format describes the behavior of a system with the help of *synchronous guarded actions* and *default reactions*. Thereby, guarded actions describe how values of variables are actively determined and the default reactions serve as a fallback, i. e. they determine the value of a variable when no guarded action does so.

Synchronous Guarded Actions

Synchronous guarded actions are used to describe the system's behavior by assignments that are executed to define a value of a variable under a certain condition. Thereby, a guarded action has the form:

$$\gamma \Rightarrow A$$

where γ is called the *guard* and A is an *action*. For the actions, only *immediate* and *delayed assignments* are considered in this thesis. The complete intermediate format also contains other kinds of actions such as *assumptions* or *assertions*, which are mainly used for verification purposes. The intention of a guarded action is that the action is executed in each instant in which its guard evaluates to **true**. Since the intermediate format also follows the synchronous abstraction, all guarded actions are evaluated in an instant synchronously in zero time and if the guard γ of an immediate assignment $\gamma \Rightarrow x = \tau$ is **true**, the right-hand side τ is evaluated to determine the value of the variable x . In contrast, delayed assignments affect the value of the next step of a variable. The assignments which occur in a QUARTZ program can be directly translated to guarded actions. However, not only the data flow is encoded in guarded actions but also the control flow. Therefore, all program labels and also the implicit start label `st` are encoded as Boolean events. The control flow can then be described by delayed actions of the form $\gamma \Rightarrow \text{next}(\ell) = \text{true}$, where γ is a condition that is responsible for moving the control flow at the next point of time to location ℓ .

The guarded actions of the program `M`, which is shown in Figure 2.6 (a), are given in Figure 2.13. For example, the data-flow action $l1 \wedge a > 4 \Rightarrow y = b$ is executed when the control flow is at label `l1` *and* the condition of the `if` statement holds. The guarded actions for the control flow are also rather simple. They express under which condition from a `pause` statement the next `pause` statement is reached. For example, if a macro step starts from label `l2`, it will end at the `pause` statement with label `l3` as expressed by the guarded action $l2 \Rightarrow \text{next}(l3) = \text{true}$. Hence, `l3` will hold in the following step.

$\begin{aligned} \text{st} &\Rightarrow x = a \\ \text{st} &\Rightarrow \text{next}(z) = b \\ l1 &\Rightarrow x = y + z \\ l1 \wedge a > 4 &\Rightarrow y = b \\ l3 &\Rightarrow z = 3 \\ l3 &\Rightarrow x = a \\ l3 &\Rightarrow \text{next}(z) = b \end{aligned}$	$\begin{aligned} \text{st} &\Rightarrow \text{next}(l1) = \text{true} \\ l1 \wedge a > 4 &\Rightarrow \text{next}(l2) = \text{true} \\ l1 \wedge \neg(a > 4) &\Rightarrow \text{next}(l3) = \text{true} \\ l2 &\Rightarrow \text{next}(l3) = \text{true} \\ l3 &\Rightarrow \text{next}(l1) = \text{true} \end{aligned}$
(a) <i>Data-Flow Actions</i>	(b) <i>Control-Flow Actions</i>

Fig. 2.13. Guarded Actions of Program `M`

Although the example illustrates the mapping from the source code to guarded actions, the guarded actions of the example `ABRO` shown in Figure 2.14 are more interesting. The source code has already been given in Figure 2.7 (a). There exists only one data-flow action which sets the output `o`. The guard of this action describes the conditions when the parallel statement terminates and the execution is not aborted by the input signal `r`. The parallel statement terminates when both, `a` and `b` have occurred. The guard can also be found in the EFSM of the `ABRO` example in Figure 2.7. It is the disjunction of the conditions of all transitions which go to the state $\{w_r\}$, because those are exactly the transitions which set `o`.

$$\begin{array}{l}
\neg r \wedge \left(\begin{array}{l} wa \wedge a \wedge \neg wb \vee \\ wb \wedge b \wedge \neg wa \vee \\ wa \wedge a \wedge wb \wedge b \end{array} \right) \Rightarrow o = \mathbf{true} \\
\end{array}
\qquad
\begin{array}{l}
st \Rightarrow \text{next}(wa) = \mathbf{true} \\
st \Rightarrow \text{next}(wb) = \mathbf{true} \\
\neg r \wedge wa \wedge \neg a \Rightarrow \text{next}(wa) = \mathbf{true} \\
\neg r \wedge wb \wedge \neg b \Rightarrow \text{next}(wb) = \mathbf{true} \\
r \wedge (wr \vee wa \vee wb) \Rightarrow \text{next}(wa) = \mathbf{true} \\
r \wedge (wr \vee wa \vee wb) \Rightarrow \text{next}(wb) = \mathbf{true} \\
\neg r \wedge \left(\begin{array}{l} wa \wedge a \wedge \neg wb \vee \\ wb \wedge b \wedge \neg wa \vee \\ wa \wedge a \wedge wb \wedge b \end{array} \right) \Rightarrow \text{next}(wr) = \mathbf{true} \\
\end{array}$$

(a) Data-Flow Actions
(b) Control-Flow Actions

Fig. 2.14. Guarded Actions of Program ABRO

Default Reactions

Similar to QUARTZ programs, the AIF description adds an implicit *default reaction* for each variable: if no guarded action determines the value for a variable, then a variable either gets a default value or stores its previous value, depending on the declaration of the variable. Obviously, this is the case if the guards of all immediate assignments in the current step and the guards of all delayed assignments in the preceding step of a variable are evaluated to **false**. Thereby, event variables are reset to a default value while memorized variables keep their value of the previous step. The default reaction is either implicitly given by the storage type of the variable or it is explicitly set. Thereby, for a variable x , it is defined by two expressions, $\text{default}^0(x)$ and $\text{default}^+(x)$. $\text{default}^0(x)$ defines the default value of x in the initial step and $\text{default}^+(x)$ defines the default value of x in each following step. Thereby, $\text{default}^+(x)$ is evaluated in the previous step, hence, it is used to *transfer* a value from the previous step to the current one. The implicitly given initial and transition value of a memorized or event variable x is defined by:

$$\begin{array}{ll}
\text{default}^0(x) := \text{default}(x) & \text{default}^0(x) := \text{default}(x) \\
\text{default}^+(x) := x & \text{default}^+(x) := \text{default}(x) \\
\textit{Memorized Variable} & \textit{Event Variable}
\end{array}$$

Thereby, $\text{default}(x)$ is the default value of the variable x which is defined by the type of x . These are the implicit default reactions which are given to a variable based on its declaration. AIF allows to explicitly overwrite the default reaction for a variable, which is used for a correct translation of local declarations and efficient handling of schizophrenia. Thereby, the default reaction is used to set a variable to the correct value when its scope is re-entered.

2.2.6 Compilation

The compilation algorithm [BrSc11a, BrSc09] translates a given QUARTZ program to the intermediate format and it is based on two things: First, the distinction of *surface* and *depth* [Berr99] of each statement and second, the computation of some *control-flow predicates* [Schn00, Schn01a].

Surface and Depth

The compilation algorithm works recursively on the AST of a QUARTZ program. Thereby, it determines the surface and depth of each (sub-)statement:

- *Surface*
The surface of a statement contains the guarded actions which are executed in the macro step in which the statement is started. These are all assignments which are possibly executed before the first `pause` statement is reached.
- *Depth*
The depth of a statement are the guarded actions which are executed in any later step, thus when the control flow is at a label inside the statement.

The relationship of surface and depth of a statement is illustrated in Figure 2.15. It shows a short program fragment together with its control-flow graph at the left side. The `if` statement leads to a branch, the small dots indicate an action and the big dots indicate a `pause` statement. The surface of this whole fragment is obtained by traversing the control-flow graph until a `pause` statement is reached. The depth can be obtained by starting at each `pause` statement and traversing the graph to the end. The sub-graphs for surface and depth are shown at the right-hand side. Generally, the depth can have multiple entry points. It can be also seen that surface and depth generally overlap: the assignment to variable `z` belongs to surface *and* the depth in the example.

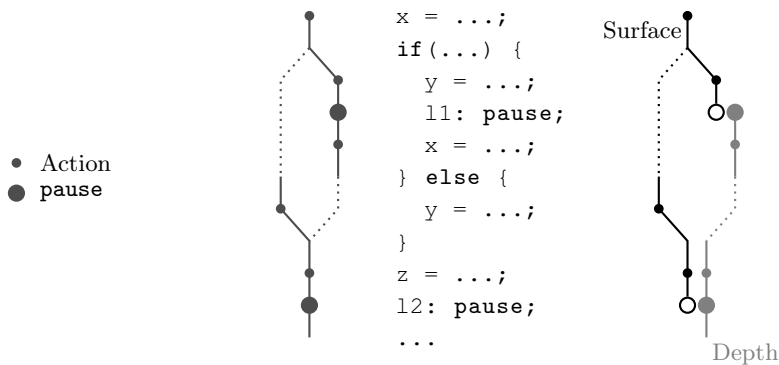


Fig. 2.15. Surface and Depth of Statements

This distinction is the key to compile complex statements of imperative synchronous languages such as local variables, abortion and suspension statements. All of them have in common that the behavior of the surface and depth must be treated separately. For instance, a schizophrenic local variable is handled in QUARTZ by duplicating the surface of its scope and, a usual abortion statement affects the execution when the statement has been entered. In terms of surface and depth, abortion takes place in the depth but not in the surface. Thus, it can be seen that this separation is reasonable to translate the behavior of those statements.

Control-Flow Predicates

The second key aspect of the compilation algorithm is the computation of the control-flow predicates of each (sub-)statement. Thereby, some of the predicates are determined by the context of a statement, while others are defined by the statement itself. The following predicates are defined by the context:

- strt_S
This condition holds when a statement is started, i. e. when the position of the statement is reached in source code. It is used to build the guards of the guarded actions.
- abrt_S
The predicate summarizes the abortion conditions of the surrounding `abort` blocks. The execution at this statement is aborted, when at least one of the surrounding abortion conditions hold.
- susp_S
Similar to abrt_S , this condition summarizes the conditions of the surrounding suspension blocks. The priority of abortion and suspension is respected by those conditions.
- strg_S
The strong and weak variants of preemption statements differ in the way, the data flow is handled. This condition holds, when the data flow is not executed, i. e. when it is either aborted or suspended.

It can be easily seen that those conditions are determined from the context where a statement is used. As already stated above, there are also conditions which are defined by a statement itself. For example, the sequence of two statements needs the information when the first one is finished, because this condition is required to define the strt_S condition of the second statement. The following predicates are defined:

- insd_S
The predicate holds when the control flow is inside the statement. This condition is a disjunction of all labels which occur in the statement.
- term_S
This condition holds when a statement is active and it terminates on its own, i. e. it is not forced to terminate. Hence, abortion conditions from the context are not taken into account.
- inst_S
This condition holds when the execution of the statement is finished in the same instant where it is started.

With the control-flow predicates and with the definition of surface and depth, the compilation algorithm can translate a program to guarded actions.

Compilation

The compilation algorithm works recursively on the AST of a QUARTZ program. Thereby it simply determines surface and depth of the whole program which together define the

guarded actions. Thereby, the guarded actions of the surface are additionally strengthened by the implicit start label `st`. As already mentioned, schizophrenic local variables are handled by the compiler by duplicating the surface of loop bodies and renaming the local variables inside this surface. The surface of the loop body is the part which can interfere with the old scope of the local declaration. Additionally, delayed assignments which are executed when a loop body is left are disabled by an additional condition that they cannot affect the local variable anymore. The second issue of synchronous languages is causality. The QUARTZ compiler translates the programs to guarded actions and *preserves* the constructivity. Hence, a program is constructive, iff its AIF representation is.

2.2.7 Code Generation

The intermediate representation based on guarded actions is a good starting point for code generation to various targets. The complexity from the source language is reduced, but the synchronous semantics is kept. Different target code generators rely on this simplified representation to translate it to other languages. Code generation for QUARTZ (as it is implemented by AVEREST) targets software, hardware and verification in different languages and models. Software can be generated from AIF in a sequential manner for single-threaded execution. However, software can also be generated for parallel [BaBS10, BaBS10a] and for distributed [BaBS11b, BaBS12] architectures. Hardware is generated by translating AIF to VERILOG. Thereby, a synchronous hardware description is generated: each clock cycle of the hardware coincides with one step in the source code. For any target language, the type system of QUARTZ also needs to be translated: the data types and the operators have to be mapped to the target language. This thesis, however, focuses on the general translation of the computational model and so it goes not into detail in translating the type system. In the following, code generation for sequential software and for hardware is illustrated.

Hardware

The translation from AIF to hardware is based on synchronous circuits, whereas each clock cycle of the resulting circuit coincides with one execution step of the original program. This translation sounds reasonable since synchronous circuits rely on the same computational model. However, this is not a must and a translation to asynchronous circuits is also possible. An interesting hardware implementation of ESTEREL is proposed in [Berr91] using a structural translation based on the source code: each statement in the source code is translated to a hardware block and the wires are connected accordingly to the program structure. Those control wires are similar to the control signals used in the QUARTZ compiler.

The generated hardware for QUARTZ is represented by equations which then can be easily translated to a hardware description language like VERILOG. An equation defines either the value of a *wire* or the value of a *register*. The values of wires can be directly computed in the current clock cycle, and they are represented by an equation of the form

$$x = \tau$$

meaning that a variable x set to the current value of the expression τ . Registers are updated during a clock transition at the end of a clock cycle. Thus, they can be used to model the

behavior of delayed assignments. A register is represented by two equations: one defines the value for the initial clock cycle, and the other one defines the transition to the following clock cycles:

$$\begin{aligned}\text{init}(x) &= \tau_1 \\ \text{next}(x) &= \tau_2\end{aligned}$$

Since those equations can be easily mapped to a hardware description language such as VERILOG, for further processing, existing tools for these languages can be used.

Control Flow

The control flow is a special case and it is considered first. The actions of the control flow sets the labels **true** when their guards hold. Each label can be separately translated to equations. Assume that the label ℓ is written by the following actions:

$$\begin{aligned}\gamma_1 &\Rightarrow \text{next}(\ell) = \mathbf{true} \\ \gamma_2 &\Rightarrow \text{next}(\ell) = \mathbf{true} \\ &\dots \\ \gamma_n &\Rightarrow \text{next}(\ell) = \mathbf{true}\end{aligned}$$

The label can then be set by these guarded actions for the next step. It will remain active for only one clock cycle, if it is not set again by a guarded action. Therefore, the actions are combined to define a register in the following way:

$$\begin{aligned}\text{init}(\ell) &= \mathbf{false} \\ \text{next}(\ell) &= \gamma_1 \vee \gamma_2 \vee \dots \vee \gamma_n\end{aligned}$$

This sets the label only when one of the guards was true in the preceding clock cycle. Since labels are considered as events, they automatically reset by the default reaction when they are not explicitly set. This is also covered in this definition. Another special case is given by the implicit start label **st** which is set to **true** for the first step and remains **false** for all other steps. The register is defined by:

$$\begin{aligned}\text{init}(\mathbf{st}) &= \mathbf{true} \\ \text{next}(\mathbf{st}) &= \mathbf{false}\end{aligned}$$

It is set initially and reset for the rest of the execution.

Data Flow

The translation of the data flow is more sophisticated because data-flow variables can be written by delayed *and* immediate assignments. Since the same variable cannot be used as a register and a wire, a new one has to be introduced. Assume that a variable x is written by the following guarded actions:

$$\begin{array}{ll}\gamma_1^i \Rightarrow x = \tau_1^i & \gamma_1^d \Rightarrow \text{next}(x) = \tau_1^d \\ \gamma_2^i \Rightarrow x = \tau_2^i & \gamma_2^d \Rightarrow \text{next}(x) = \tau_2^d \\ \dots & \dots \\ \gamma_n^i \Rightarrow x = \tau_n^i & \gamma_m^d \Rightarrow \text{next}(x) = \tau_m^d\end{array}$$

A new identifier x^{next} is introduced for the variable x and is used to define a register to store the values from delayed assignments. The equations can then be defined as follows:

$$\begin{aligned} \text{init}(x^{\text{next}}) &= \text{default}^0(x) \\ \text{next}(x^{\text{next}}) &= \begin{cases} \tau_1^d & , \text{if } \gamma_1^d \\ \tau_2^d & , \text{if } \gamma_2^d \\ \vdots & \vdots \\ \tau_m^d & , \text{if } \gamma_m^d \\ \text{default}^+(x) & , \text{else} \end{cases} \quad x = \begin{cases} \tau_1^i & , \text{if } \gamma_1^i \\ \tau_2^i & , \text{if } \gamma_2^i \\ \vdots & \vdots \\ \tau_m^i & , \text{if } \gamma_m^i \\ x^{\text{next}} & , \text{else} \end{cases} \end{aligned}$$

Thus, x gets its value from an immediate assignment whenever one of the guards holds. Otherwise, the value of x^{next} is used which is either a value from a delayed assignment of the preceding step, or the value of the default reaction if no delayed assignment issued a value. Thus, the value from the default reaction is used when no delayed assignment of the previous step and no immediate assignment of the current step sets a value.

This translation considers the general translation for sake of completeness. For special cases, the translation can be much simpler. For example, for event variables which are not set by delayed assignments, the additional register can be omitted, because

$$\text{default}^0(x) = \text{default}^+(x) = \text{default}(x)$$

holds, and the register would hold the value $\text{default}(x)$ every time. Thus, x^{next} can simply be substituted by this value.

Example

An example is given by the equations for the variable z from the QUARTZ program `M` whose guarded actions are given in Figure 2.13. The resulting equations for this variable are:

$$\begin{aligned} z &= \begin{cases} 3 & , \text{if } l3 \\ z^{\text{next}} & , \text{else} \end{cases} \\ \text{init}(z^{\text{next}}) &= 0 \\ \text{next}(z^{\text{next}}) &= \begin{cases} b & , \text{if } st \\ b & , \text{if } l3 \\ z & , \text{else} \end{cases} \end{aligned}$$

Since z is a memorized variable, $\text{default}^+(z) = z$ holds. This is different to the variable \circ of the QUARTZ program `ABRO` (Figure 2.14) which is declared as an event variable. The equations for \circ are:

$$\begin{aligned} \circ &= \begin{cases} \text{true} & , \text{if } \begin{pmatrix} wa \wedge a \wedge \neg wb & \vee \\ wb \wedge b \wedge \neg wa & \vee \\ wa \wedge a \wedge wb \wedge b \end{pmatrix} \\ \circ^{\text{next}} & , \text{else} \end{cases} \\ \text{init}(\circ^{\text{next}}) &= \text{false} \\ \text{next}(\circ^{\text{next}}) &= \text{false} \end{aligned}$$

It also illustrates the already mentioned optimization. The additional register \circ^{next} is not needed, because it would hold the value `false` every time. Thus, it can be omitted.

Software

The translation of AIF to sequential software is considered in this section. The semantics of AIF can be given by a fixpoint iteration over all guarded actions to compute the values for a step. This is reasonable for a hardware translation, because the fixpoint iteration is implicitly done by the synchronous circuit. However, there are more efficient solutions for a translation to sequential software. Therefore, the translation tries to order the guarded actions that only one iteration is needed to compute all values. It is required that all used variables of a guarded action have been computed before its execution. Thus, if a guarded action writes the variable x it must be executed before all guarded actions which read variable x .

Basic Scheme

The basic idea of sequential code generation is illustrated by the code stub in Figure 2.16. First, declaration and initialization of the variables is done. Then, a loop repeats the evaluation of the guarded actions, where each iteration is the execution of a macro step. At the beginning of an iteration, all inputs are read from the environment. With the input values, the (immediate) guarded actions can be evaluated to determine the value of each variable. Thereby, the actions have to be ordered that all actions reading a variable x are executed after all actions possibly write x to ensure that x is determined before it is used. After all immediate actions are executed, the values of all variables of a step are determined and the outputs can be written to the environment. The following step is then prepared by executing the delayed guarded actions which are evaluated with the current values but set the variables for the next step. The delayed actions have to be ordered in the opposite direction here: all actions reading a variable x must be executed before a delayed assignment sets a new value to x , because otherwise the value of the next step of x would be read.

```

// initialize
...
while(1) {
    // read all inputs
    ...
    // evaluate all immediate assignments
    ...
    // write all outputs
    ...
    // evaluate all delayed assignments
    ...
}

```

Fig. 2.16. Sequential Code Stub

The default values of variables are not yet covered by the presented scheme. Recall that a default value is assigned to a variable for a macro step, when it is not set by a guarded action. For event variables, the default value of the according type is assigned, for memorized

variables the value from the preceding step is used. Since the variables keep their values in sequential code, for this second case no additional effort is necessary. However, events have to be reset when they are not explicitly set. Therefore, event variables can be reset at the beginning of a step and they would be either overwritten by an assignment or the implicit default value would remain.

Cyclic Dependencies

The above description can generally lead to cycles in the dependencies of guarded actions. Thereby, it has to be distinguished whether the cycle occurs due to the read-after-write dependencies between immediate guarded actions or due to the write-after-read dependencies between delayed guarded actions. Cyclic dependencies between immediate guarded actions can be the result of a causality problem, then there is no way to handle them. However, they can be resolved by either eliminating the dependencies when the guards of the actions exclude each other, or by a deeper analysis. A way to do this can be found in [Edwa03a].

Cyclic dependencies between delayed guarded actions are not a causality problem, because in the original model they write values for the following step. The dependencies are introduced by the fact that the same variable is used for the current and the following value in the implementation. In this case, it is possible to break the cycle by introducing a new (temporary) variable which serves as a placeholder for the value for the following step. Thus, for a delayed assignment, instead of writing the variable x , it writes a new variable x^{next} and the value of x is still valid. At the end of the loop iteration, the value have to be written back to x .

EFSM-based Code Generation

The code generation for a set of guarded actions has been considered so far. Thereby, all guarded actions are evaluated in each loop iteration. However, the number of guarded actions which are potentially executed within a step can easily be reduced by first considering the labels. The EFSM which already has been introduced for the QUARTZ example ABRO in Figure 2.8 enumerates the reachable label combinations, the states. According to those states, a partial evaluation of the labels can reduce the number of guarded actions which has to be considered in each state. The above introduced code generation can now be applied to each state and an `if` statement can filter for the state first. This can reduce the number of guarded actions which are considered within an iteration drastically and the generated code is very fast. However, since the size of the EFSM is generally exponential w. r. t. the original program, also the size of the generated code may be unmanageable. On the other hand, this shows that there is a choice between efficiency and small code.

Example

An example of sequential code for the QUARTZ program M is given in Figure 2.17. The generated code is based on the EFSM for M and only the state which relates to the label `l1` is shown. The immediate assignments are ordered: the assignment to `y` is executed before the assignment to `x`. The control flow of the EFSM is encoded in the variable `__state000` which is assigned by a delayed assignment: the state is updated for the next iteration of the loop. In the example, the next state depends on the input `a`.

```

void M() {
    // initialize
    ...
    while(1) {
        switch(__state000) {
            ...
            case 0x1: {
                // read all inputs
                READ_i1(a);
                READ_i2(b);
                // evaluate all immediate assignments
                if(a > 4)
                    y = b;
                x = (y + z);
                // write all outputs
                WRITE_x(x);
                WRITE_y(y);
                // evaluate all delayed assignments
                if(a > 4)
                    __state000 = 0x2;
                if(! (a > 4))
                    __state000 = 0x3;
            }
            break;
            ...
        }
    }
}

```

Fig. 2.17. Sequential C-Code for QUARTZ Program M

2.2.8 AVEREST

The complete design flow illustrated in Figure 2.18 for the synchronous languages QUARTZ is implemented with the help of the AVEREST library [AVEREST]. Thereby, the toolchain also implements modular compilation which is based on a special intermediate format, which is called AIF Module: each QUARTZ module is translated to an appropriate AIF, and a linker composes these to an AIF system. Therefore, AIF modules stores additional information of calling contexts and called modules, which are used for the linker. AIF systems represent the intermediate format which has been described in Section 2.2.6. Code generation is then based on AIF systems where some special transformations have been applied beforehand. For example, the generation of the EFSM for software or the generation of an equation system for hardware are implemented as a transformation. Then, backend tools are used to write the source files in the various target languages.

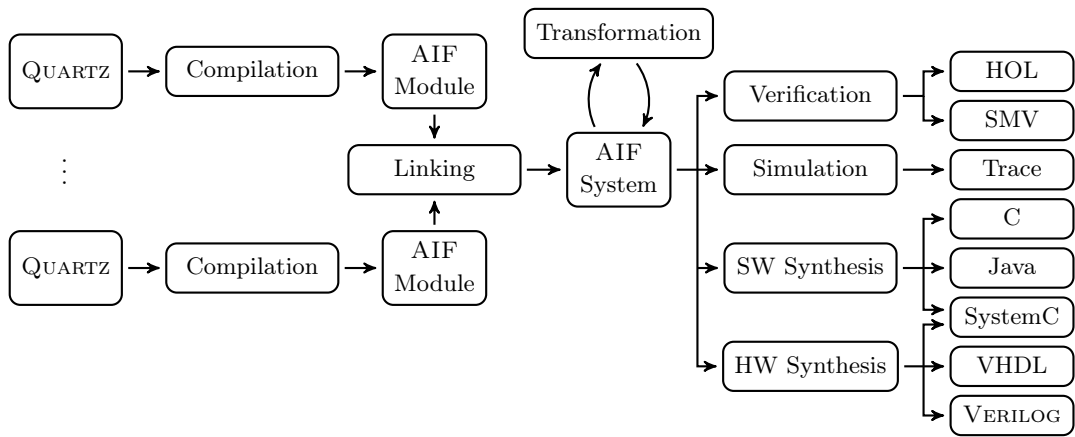


Fig. 2.18. QUARTZ Design Flow

Chapter 3

Clock Refinement

The main contribution of this thesis, the clock refinement extension to the imperative synchronous language QUARTZ, is informally introduced in this chapter. To this end, some drawbacks and limitations of the current state of imperative synchronous languages are pointed out, followed by an introduction of clock refinement with the help of a simple motivating example. After the basic idea has been explained, the impact of the extension to the existing QUARTZ statements and its behavior is discussed. For example, the extended behavior of abortion, suspension, and delayed assignments are considered in detail. Afterwards, the discussion about constructivity and logical correctness that was started for QUARTZ is also adapted for the extension. In the rest of this thesis, the terms *pure* QUARTZ and *single-clock* QUARTZ are used for the original version of QUARTZ, and the terms *extension* and *extended* QUARTZ are used for the clock refinement extension. The extension was first introduced in [GeBS10] and an extended overview can be found in [GeBS13].

3.1 Limitations of QUARTZ

The synchronous abstraction idealizes the view to systems by reaction instants in which outputs are immediately produced when inputs arrive. It leads to a simpler system description and simplifies composition, modelling and analysis of the systems [BeBe91]. As explained in Section 2.2, this model is implemented by the synchronous language QUARTZ by `pause` statements identifying the end of a reaction: all actions executed until the next `pause` statements are reached, contribute to the current reaction. As a consequence, parallel threads run in lockstep and execute each reaction synchronously based on the common global variable environment. The fact that variables only have one value per reaction, is a key feature to make parallel execution and also abortion and suspension deterministic. However, this abstraction imposes a single time scale: in each reaction all inputs are read and all outputs are produced. There is no means to express independent execution or more flexibility in timing. Some limitations imposed by this single abstraction are discussed in the following.

- Currently, QUARTZ offers module calls as a feature, but they behave in a fully synchronous manner: each step of the calling module is executed synchronously with a step of the called module. A common feature offered by many programming languages are *functions* taking some input values, performing a computation and providing output values. In

QUARTZ, only very simple functions can be represented by macros of expressions, but they cannot represent complex computations. The language ESTEREL allows to integrate programs of a *host language*. However, since the host language is not covered by the tools, it is not available for simulation, analyses and verification, and also synthesis is no longer flexibly possible to all target languages. A better solution would be to integrate a mechanism in the language itself so that functions, or at least modules that can behave like functions, can be directly implemented in the synchronous language.

- Deterministic concurrency is enforced by the synchronous model as it is implemented in QUARTZ, but this imposes restrictions on the modelling possibilities and also on code generation, since threads synchronize their execution at each `pause` statement even if the threads do not communicate with each other. While a static analysis may be able to detect the dependencies to desynchronize such programs, another possibility is to add an explicit notion of independence to the language which can be used by the programmer to express the independence of threads in certain program locations. In this way, the compiler can create desynchronized code without sophisticated and expensive analyses.
- Having only one temporal abstraction layer can make a system description inflexible with respect to temporal changes. If a component is exchanged by another implementation having the same functional behavior but a different number of computation steps, the behavior of the whole system can be changed. Anyway, it is in general not an insolvable task to write programs in a way that they are stable for such changes, but a better solution would be to integrate features in the language allowing the programmer to express this kind of communication explicitly.

The solution proposed in this thesis to these limitations is a language extension to QUARTZ introduced in the following section. However, keep in mind that the extension does not extend the expressiveness of the language in general, but it allows the programmer to express things in a more convenient and liberal way which finally also allows the compiler to generate more flexible code.

3.2 Basic Idea of Clock Refinement

The aim of the extension is to reduce the limitations coming from the single time scale by introducing hierarchical synchronous layers which by themselves follow the synchronous abstraction. Practically, QUARTZ is enriched with the possibility to divide a macro step by *smaller substeps* of a lower layer that can be seen as macro steps on their own level. Hence, the basic notion of steps is kept, but it can be abstracted in the hierarchy. The excessive need for synchronization which was addressed before can be avoided by substeps which refine existing steps and do not need to synchronize. In pure QUARTZ, no explicit clock is given, but it can be seen as a single-clock model where each step (instant) belongs to a clock tick. In this interpretation, `pause` statements are barriers for the next clock tick. Therefore, it seems to be natural to keep this notion and identify the substeps also with new (local) clocks. This section explains the introduction of this concept by an example showing the new statements which are introduced to define new clocks and to use them to define substeps. It also shows how one of the presented limitations of QUARTZ, namely the implementation of functions, are addressed.

The idea is illustrated by the two implementations of the *Euclidean Algorithm* given in Figure 3.1. The first variant (given on the left-hand side of the figure) is implemented in pure QUARTZ without clock refinement. The module reads its two inputs a and b in the first step and assigns them to the local variables x and y used in the loop to compute the *Greatest Common Divisor (GCD)*. The iteration steps of the loop are separated by the `pause` statement with label `l`. Each variable has a unique value in a step, and the delayed assignments set a new value to the variables for the following step. Finally, the computed GCD is written to the output variable `gcd`. The apparent drawback has been discussed above: the computation needs several steps, and the number of steps required depends on the input values, and each call to the module has to take care of the time consumption. Changing the algorithm, e.g. to use the `mod` operator instead of subtraction, would also change the time consumption possibly also affecting the behavior of the whole system. An example execution trace for the computation of the GCD of the numbers 7 and 3 is shown in Figure 3.2 (a). The computation takes 6 steps and during this computation, the inputs a and b may change in general. Thus, a calling module has to take care of the computation steps until the result is available. Please note also that it is not possible to compute the GCD within one step, because the number of needed iterations depends on the input values, and the number of actions in a step must be statically bounded.

<pre> module GCD1(nat ?a, ?b, !gcd) { nat x, y; x = a; y = b; while(x > 0) { if(x >= y) next(x) = x-y; else next(y) = y-x; l: pause; } gcd = y; } </pre>	<pre> module GCD2(nat ?a, ?b, !gcd) { clock(C1) { nat x, y; x = a; y = b; while(x > 0) { if(x >= y) next(x) = x-y; else next(y) = y-x; l: pause(C1); } gcd = y; } } </pre>
(a) <i>Single Clock</i>	(b) <i>Clock Refinement</i>

Fig. 3.1. Greatest Common Divisor

The second variant using clock refinement is shown on the right hand side of Figure 3.1. While the overall algorithm remains the same, the GCD computation is now hidden in the declaration of the local clock `C1`. The computation steps are separated by the `pause` statement with label `l` now belonging to the clock `C1`. In contrast to the first variant, the computation does not hit a `pause` statement of the outer clock and thus, the computation steps are not visible to the outside: the computation is done in substeps of a macro step of

the module. As a consequence, each call to this module appears to be completed in a single step. The local variables x and y are now declared in the local clock block and therefore, they can change their value for each step of the local clock, which is crucial for the correct execution of the algorithm in this example. A trace for the computation of the GCD of the numbers 7 and 3 is shown in Figure 3.2 (b). The computation for this version takes also 6 steps, but these are steps of clock $C1$, whereas the computation is finished in one step of the module clock. The variables a , b and gcd , which are declared on the module clock, only have one value for this base step, while the variables x and y , which are declared on clock $C1$, change their value for each step of clock $C1$. Thus, the inputs remain constant during the computation, and there is only one value of the output gcd .

The trace shows even more: In the synchronous model, each variable has exactly one value for each step, and this value is valid from the beginning to the end of the step. The inputs are given from the outside and thus, they are known for the whole computation. The output gcd is computed after some substeps, but in the general view, it is valid during the whole step. This point will be considered later in the context of constructivity.

		1	2	3	4	5	6
a		7	7	7	7	7	7
b		3	3	3	3	3	3
st		T	F	F	F	F	F
l		F	T	T	T	T	T
x		7	4	1	1	1	0
y		3	3	3	2	1	1
gcd		0	0	0	0	0	1

(a) *Single Clock*

		1	2	3	4	5	6	
a		7	—————					
b		3	—————					
st		T	F	F	F	F	F	
l		F	T	T	T	T	T	
x		7	4	1	1	1	0	
y		3	3	3	2	1	1	
gcd		—————						1

(b) *Clock Refinement*

Fig. 3.2. Greatest Common Divisor Traces

The introduction of the local clock declaration enables the division of a macro step of a module by *smaller* steps which are associated to the new clock. In the following, the clock of the module (which the module steps are based on) is implicitly given by $C0$. The `pause` statements related to this clock do not explicitly need the clock annotation, i. e. `pause` \equiv `pause (C0)`. An illustration of the relation of steps and substeps is given in Figure 3.3. The trace at the upper side shows a sequence of macro steps of a pure QUARTZ program. The communication with the environment is done in each reaction. Since the macro steps are based on logical time, the physical time of the steps may differ. At the lower side of the figure, an other trace of an extended QUARTZ program is shown. Thereby, the clock of the module is refined by the clock $C1$, which is by itself refined by the clock $C2$. Thus, a step of the module can be divided by smaller steps related to clock $C1$ and those smaller steps can be divided by even smaller steps of the clock $C2$. According to the logical time scale of macro steps, the division of steps into substeps is also logically defined, and there is not necessarily a fixed number of substeps.

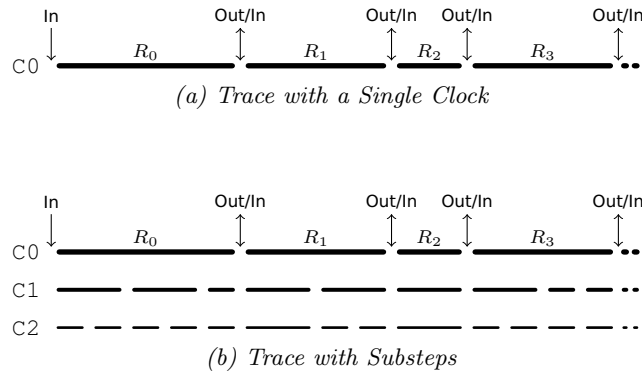


Fig. 3.3. Refinement of Steps

Obviously, it is not only possible to arbitrarily nest clock declarations but also to introduce new clocks in separate scopes. This gives rise to the clock tree of a program, which can be obtained from the program structure. Figure 3.4 shows an example: the left-hand side shows the structure of nested clock declarations in source code, and the right-hand side shows the corresponding clock tree, which can be directly derived from it. The terms *higher* and *lower* are used to classify the order of clocks on a branch of the tree. Hence, the clock C0 is the highest clock of a module. Furthermore, the terms *unrelated* and *independent* are used for clocks on different branches, e.g. the clocks C4 and C1 are unrelated in the example. Furthermore, at each position in the source code, only one branch of the clocks is visible, and unrelated clocks can never be visible at the same point.

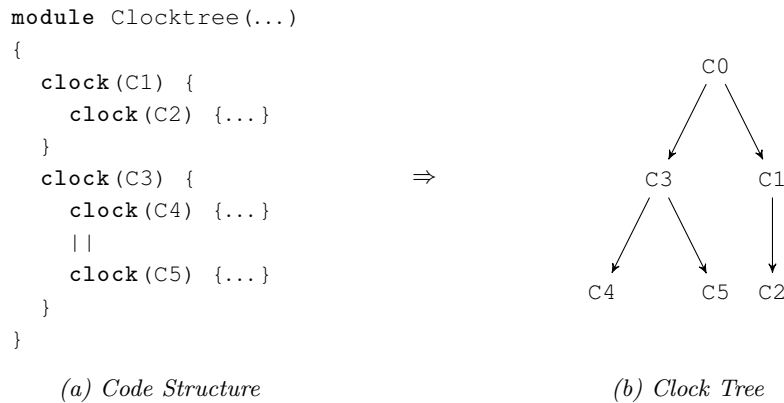


Fig. 3.4. Clock Tree of Program

3.2.1 Steps, Variables and Assignments

The advantage that variables can now be declared for refined clocks and change their value in each substep was already pointed out by the GCD example. The clock of a variable is

thereby implicitly given by the least visible clock at the point of the declaration. Hence, the variable is only visible when its clock is, and like for clocks, it can never be the case that two variables with unrelated clocks are visible at the same source code location. The input and output variables of a module have the clock C_0 : from the point of view of the environment, the module is still a synchronous system getting one value for each input and producing one value for each output. Furthermore, since only *related* clocks are visible at each position, expressions like conditions of `if` statements or assignments can only contain variables that are related.

Values of variables can be explicitly set by either immediate or delayed assignments setting the value for the whole step of the variable's clock. Like for pure QUARTZ, immediate assignments set the value of the current step, whereas the value set by delayed assignments is transferred to next *step* of the variable's clock. However, this design decision is not obvious at the first glance, but nothing but a logical decision, since the value of a variable can only change with steps of *its* clock.

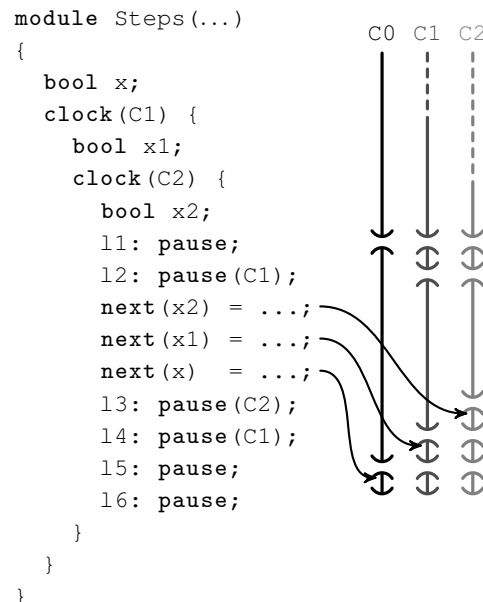


Fig. 3.5. Steps and Delayed Assignments

Figure 3.5 illustrates how values are set by delayed assignments with a code example that declares two clocks C_1 and C_2 in addition to the module clock C_0 . Thereby, the clock of variable x is C_0 , the clock of variable x_1 is C_1 , and the clock of variable x_2 is C_2 . The lines at the right-hand side of the code indicate the duration of the steps of the clocks. Thus, the variable x_1 has exactly one value between label 12 and 14. Delayed assignments set the next value of a variable, hence, e.g. the value given to x_1 by the delayed assignment is valid for the step beginning at label 14 and ending at label 15. In general, a `pause` statement of a

clock also marks the end of the steps of the lower clocks. Hence, the step between the labels 15 and 16 is a step of all clocks (C0, C1 and C2).

3.2.2 Parallel Execution

In pure QUARTZ, parallel threads run in lockstep, i. e. each step of one thread is executed together with one step of the other thread synchronously, and the parallel threads synchronize on each `pause` statement, similar to *barrier synchronization*. In the extension, the threads synchronize only by steps of a common clock and the thread can execute substeps of unrelated clocks independently. Furthermore, the variables belonging to the clocks defined in a thread are not visible to the other thread. Hence, even if they can change independently to the substeps in the other thread, they cannot be read there. Communication between the threads can only be established by variables of a common clock and for those variables the values in each step are uniquely defined.

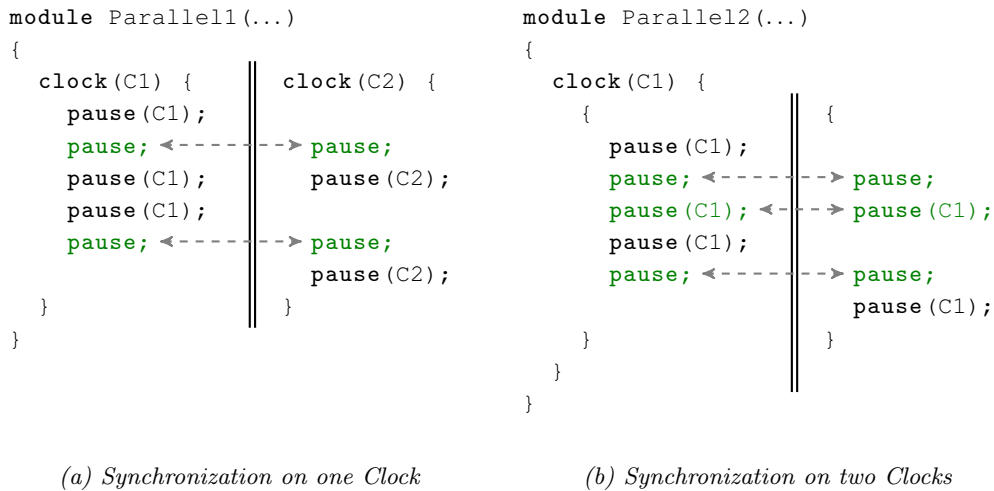


Fig. 3.6. Parallel Execution

The synchronization of parallel execution based on clocks is shown by the examples in Figure 3.6. Thereby, the threads of the parallel statement `{...} || {...}` are drawn beside each other for a better illustration of the synchronization. The first example on the left-hand side defines two new clocks, one in each thread of the parallel statement. Thus, one step of the module clock C0 is divided into different substeps in both threads, but the substeps are not related in any way. A second example for parallel synchronization is shown on the right-hand side of the figure, where only one refined clock is defined for both threads. Hence, the threads execute the steps of the module clock C0 and also the steps of the refined clock C1 together. In terms of synchronization, both threads synchronize on the `pause` statements of each common clock, which are C0 and C1 in this example. However, one step of the module clock is divided by three substeps in the first and by two substeps in the second thread. Hence, the second thread has to *wait* until the first thread also finishes

the step of clock C_0 and only the first two substeps are executed together. This is similar to the execution of parallel threads in single-clock QUARTZ: if one thread is finished but the other one is not, the first one has to wait until the execution of the second one is also finished. For refined clocks, this behavior does not only take place at the end of the whole parallel statement, but also at the end of each step of a common clock.

3.2.3 Abortion and Suspension

Another important feature of imperative synchronous languages are the abortion and suspension statements, which deterministically preempt the execution. The determinism is based on the uniquely defined interaction points: either the whole step is stopped or the whole step is executed. Obviously, the extension should also preserve the determinism as it did so far. The crucial point is to define the points of time where the preemption takes place. A preemption statement can be used within the scope of a refined clock and also new clocks can be declared within the preemption statement itself. The condition for the preemption can only be defined by variables with a higher clock because variables declared inside the preemption statement are not visible at this point. When the preemption condition holds, it will hold for the whole step and cannot be changed in substeps of a clock declared inside: either the whole step is preempted or not. According to pure QUARTZ, strong preemption takes place at the beginning and weak preemption at the end of a step. A more detailed explanation is given in the following by examples.

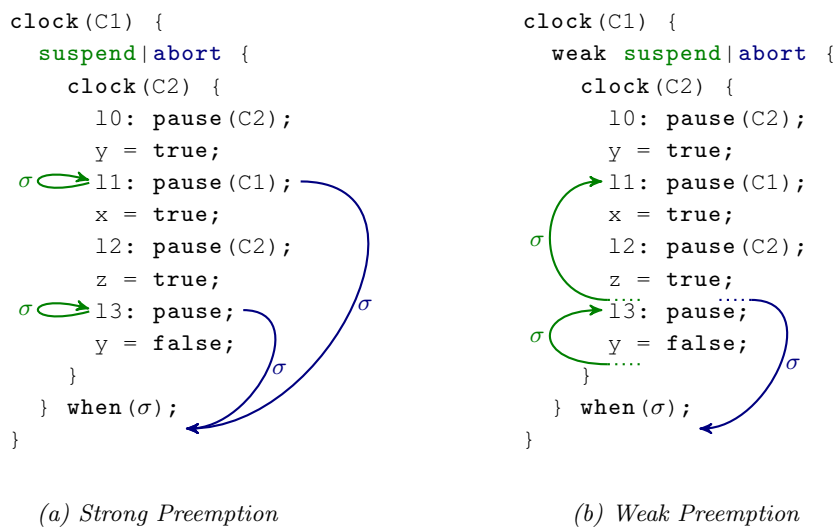


Fig. 3.7. Examples: Preemption with Refined Clocks

Figure 3.7 illustrates the weak and strong preemption in the context of refined clocks for the abortion and suspension statements both given in the same source code.

The strong abortion, which is considered first, stops the data flow and control flow of a step completely and moves the execution to the end of the abort block. The lowest clock

which is visible for the abort statement is the clock C1, and therefore, abortion operates on steps of this clock: abortion is possible at pause statements with clock C0 or C1. Thereby, according to pure QUARTZ, the abortion can only take place after the control flow is *inside* the statement, and the step entering the block is not aborted. In the example, the clock C2 is defined inside of the abort block and the entering step is divided into substeps by the clock C2. Therefore, both substeps belong to the entering step and abortion can first take place when the control flow is at label l1. The suspension behaves similarly, it can also take place from the labels l1 and l3 in the example, but due to the behavior of the suspension, nothing is executed and the control flow remains at the labels it is until the next step is started.

The figure also illustrates the weak versions of the preemption statements at the right-hand side. In contrast to the strong counterparts, preemption takes place at the end of a step. Thus, it can take place *just before* the label l3 is reached and the execution will continue after the whole abort statement. This is an interesting case because the step which is executed is divided into substeps by C2 and right before the end of the step, but after the substeps, the weak abortion takes place. The suspension also behaves similar: the step is executed, but the control-flow will remain at the label where the step was started from for the next step. For example, the step starting from l1 is divided into substeps associated to C2 and right before label l3 is reached, the execution stops (if σ holds) and remains at label l1 again where the next step will start from. Again, the substeps are executed, but finally weak suspension is done with respect to the step of clock C1.

Finally, preemption statements behave in the same way as before, but they have to take care about locally defined clocks. However, from the logical point of view, the preemption statements do not *see* the substeps, they are only aware of the steps related to clocks defined outside. Thus, they just behave like the original preemption statements with respect to the *visible* steps.

3.2.4 Determinism

After having introduced the extension and explained the basic behavior of the characteristic statements of QUARTZ, some comments shall be given on the determinism of the extension. As already said, pure QUARTZ is deterministic, because of the synchronous abstraction forcing one value per variable in a step. Based on the steps and variables, parallel threads are executed in lockstep, and also for preemption, the interaction points are well defined, since either the whole step is preempted or not. Hence, values and steps are deterministically defined and do not change due to communication or execution delays of threads.

The introduction of independent substeps in parallel threads could change this behavior when it would be possible to read a variable's values in substeps since different values could be read depending on the considered substep. But, since variables of unrelated clocks as also the clocks are not visible to the other thread, this behavior cannot occur. It turns out that whenever a variable is read, its value is uniquely defined and the independent execution cannot change this.

3.3 Constructivity vs. Logical Correctness

The previous section introduced the general behavior of the extension based on the statements of QUARTZ, whereas this section looks more into some details. The difference of *constructivity* and *logical correctness* was already pointed out for QUARTZ in Section 2.2.3. The abstraction introduced by synchronous languages combines several assignments, i. e. micro steps, to a single instant, i. e. a macro step, having each *solution*, i. e. a valuation of all variables which are consistent with the assigned values, as a possible behavior. Deterministic systems only have one possible behavior in a step and are called logical correct, but finding this unique solution can be hard for arbitrary programs. A subset of the logical correct programs are the constructive programs defined by operational rules, namely *reaction rules*, allowing to compute a solution with reasonable effort. Even more, the programs can be also translated with reasonable effort to a target language and the rules also define a *natural* way of execution, which is more accessible for developers instead of logical correctness. Generally, a constructive subset needs to be defined in a way that (1) it can be checked with reasonable effort, (2) it can be efficiently translated to target languages, and (3) it is still a *practical* and sound subset of the language. The definition of constructive programs for the extension is motivated in this section and it is formally completed in Chapter 4. To get an impression of what should be allowed in a constructive sense, first the original definition of logical correctness is generalized for the extension and examples are considered. Based on the discussion there, the notion of constructive programs for the extension is composed.

<pre> bool x, y; clock (C1) { bool z; 11: pause; if (!z); y = x; z = false; 12: pause (C1); if (z) x = true; z = true; 13: pause; } </pre>	$\mathcal{E}^{C0} = \{(x, \text{true}), (y, \text{true})\}$ $\mathcal{E}_1^{C1} = \{(z, \text{false})\}$ $\mathcal{E}_2^{C1} = \{(z, \text{true})\}$
(a) Code Example	(b) Environments

Fig. 3.8. Logical Correctness of the Extension



With the introduction of the extension, a step of a program can be divided into smaller steps of a lower clock considered as micro steps from the point of view of the higher clock. Logical correct means for pure QUARTZ that there is exactly one valuation of the variables that is consistent with the execution, but since variables of lower clocks can change their values in substeps, this does not hold anymore. A generalization leads to the condition that for each substep must exist also a unique valuation of the variables of lower clocks. An

example is given in Figure 3.8. In the code on the left-hand side, a step of the module clock C0 starts at the `pause` statement with label 11, and ends at the `pause` statement with label 13. It is divided into two substeps of clock C1 separated by the `pause` statement with label 12. The variables `x` and `y`, which are based on clock C0, have one valid value for the whole step which is reflected by the environment \mathcal{E}^{C0} on the right-hand side. The value of the variable `z`, which is based on the clock C1, can change for the substeps and is therefore illustrated by the *two* different environments \mathcal{E}_1^{C1} and \mathcal{E}_2^{C1} : it has the value `false` in the first, and the value `true` in the second substep. The presented valuation is the only valid one that is consistent with an execution of a step, hence the example is logically correct. However, the question remains if this solution can be *constructively* determined, and if so, which are the operational rules to compute it? As already said, the rules are given in a later chapter, and this section only informally discusses the possibilities. So, the question here is, if this example should be considered as a constructive program or not?

For the first substep, it is clear that the value of `z` is set to `false` by the assignment, and dependent on this value the assignment `y = x` is executed. However, it is not clear in this substep what the value of `x` is, because it is assigned in the second substep, but the value has to be the same for the whole step (of clock C0). Hence, the first substep needs the value of `x` which is determined by the second one, but the second one can only be executed after the first one. Since this example is quite simple and one might think of solving the dependencies for simulation or code generation here, it is obvious that for arbitrary programs of this type an expensive analysis is needed to transform them to constructive code. Hence, even if the presented solution is logically correct, it disagrees with the natural execution order of substeps imposed by their control flow (`x` is used before it is assigned). Furthermore, it is also natural for the programmer that the substeps are executed in this order and, since clock refinement provides several abstraction layers, it seems to be a good choice that each layer *behaves* like the original language.

3.3.1 Sequential Execution of Substeps

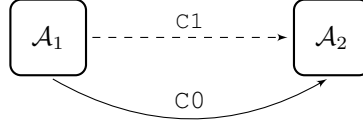
```

clock (C1) {
  10: pause;
  
  11: pause (C1);
  
  12: pause;
}

```

Fig. 3.9. Example: Sequential Execution with one Refined Clock

For the following discussion, the assignments in the source code are abstracted to *sets of actions*, which can have arbitrary dependencies between the actions. It is only required that the actions can be evaluated accordingly to constructive rules like the original reaction rules for QUARTZ: hence, cyclic dependencies must be able to be resolved with lazy evaluation as it was explained in Section 2.2.3. The source code of Figure 3.8 can be rewritten with this abstraction to the source code in Figure 3.9, where the assignments and the `if` statements that restrict the execution of the assignments are represented by the abstract sets \mathcal{A}_1 and \mathcal{A}_2 . The (constructive) execution and the dependencies are then illustrated as follows:



The actions \mathcal{A}_1 are evaluated in the first substep and the actions \mathcal{A}_2 are evaluated in the second substep. Thereby, each substep must be consistent accordingly to the particular assignments executed in the substep. The dashed line represents a sequential dependency based on substeps of clock $C1$. The solid line represents a dependency based on variables of clock $C0$ having their value for the whole step. Hence, a variable of clock $C0$ can get its value in the first substep and can be used in the second one, but the other way around is not possible due to the sequential dependency imposed by clock $C1$.

The example illustrates the order of sequential execution of substeps where a variable must be determined before it can be used. In pure QUARTZ, those dependencies only exist within such an abstract action set \mathcal{A} . This changes for the extension since e.g. variables of clock $C0$ can be written in a substep, but must have the same value for the whole step. Therefore, the dependencies of the assignments must also comply with the sequential execution order imposed by the control flow of the substeps.

3.3.2 Scheduling of Parallel Threads

The general synchronization scheme for the parallel execution of steps and substeps was explained in Section 3.2.2. The explanation there is sound and independent of any definition of constructivity. However, if constructivity is considered, there are cases where it is not directly clear what must be executed together. Unrelated clocks can be defined in different parallel threads so they can be theoretically executed independently and only have to synchronize with the `pause` statements of common clocks. But, even one refined clock in the context of parallel execution raises some questions which are considered first in this section.

Consider the example code in Figure 3.10 (a) consisting of two parallel threads inside the declaration of the clock $C1$. Both threads basically consist of one step of clock $C0$ starting from the labels 11 and 14 and ending at the labels 13 and 17. This step is divided into two substeps in the first thread and into three substeps in the second thread. Accordingly to the synchronization based on common clocks, the actions \mathcal{A}_1 , \mathcal{A}_3 are executed in the first substep, the actions \mathcal{A}_2 , \mathcal{A}_4 are executed in the second substep, and finally, the actions \mathcal{A}_5 are executed in the third substep. Hence, the first thread *reaches* the `pause` statement of clock $C0$ before the second thread and waits for it until it executed the actions \mathcal{A}_5 . Causal dependencies can be illustrated as follows:

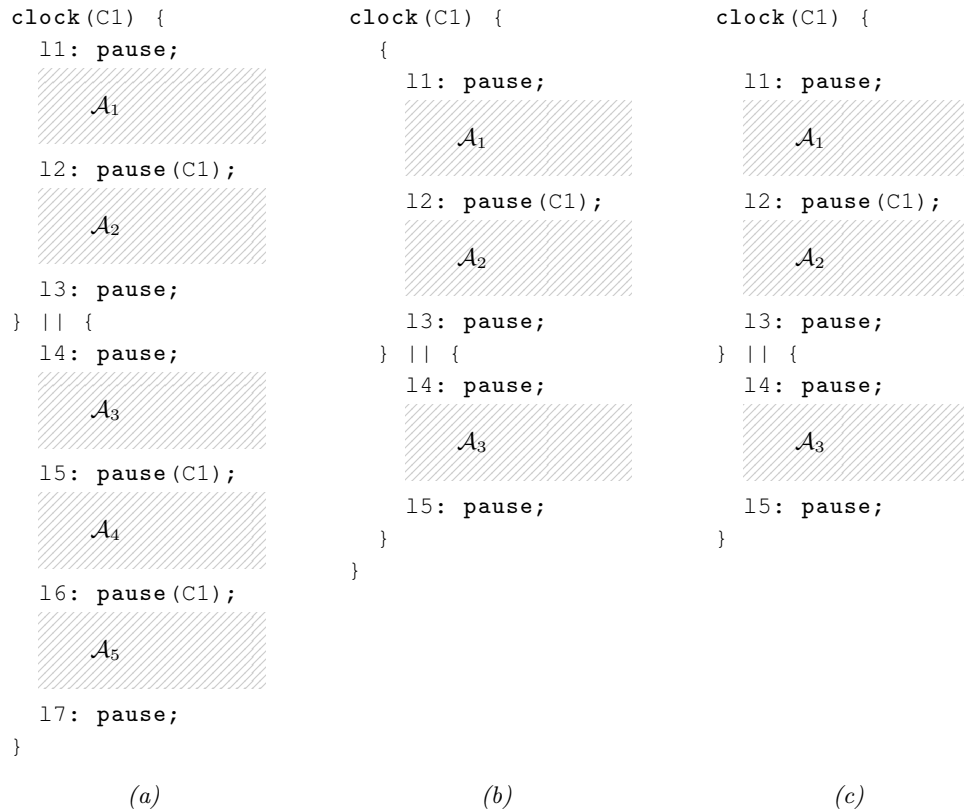
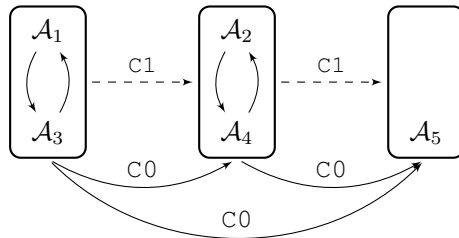


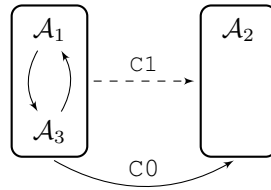
Fig. 3.10. Examples: Parallel Execution with one Refined Clock



For example in the first substep, it must be possible to resolve the dependencies between the actions \mathcal{A}_1 and \mathcal{A}_3 by constructive rules. The substeps are executed one after the other depending on the order given by the control flow. For variables of a higher clock, dependencies must follow the execution order of the substeps. For example, \mathcal{A}_2 can use variables of clock C0 which have been determined in the substep before.

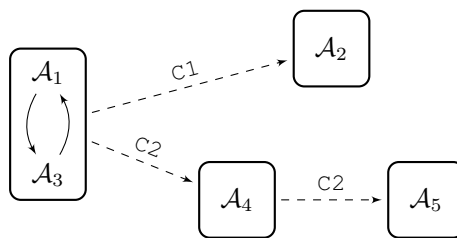
The example code given in Figure 3.10 (b) is quite similar to the one considered before, but the `pause` statements of clock C1 are omitted in the second thread, and it only consists of one step/substep. It is quite obvious that the execution scheme for this example should not really change the example before and that the actions \mathcal{A}_1 , \mathcal{A}_3 are executed in the first

substep, and the actions \mathcal{A}_2 are executed in the second one. The possible dependencies are then illustrated as:



Hence, even if the second thread does not contain any `pause` statements of clock $C1$, the step is considered as one substep. After the execution of this substep together with the first thread, it waits for the first thread until it has executed the second substep. This seems to be an obvious result, but when the third example shown in Figure 3.10 (c) is considered, it has some consequences. The difference of this third code example is that the clock $C1$ is now declared in the first thread and therefore not visible in the second one. From the point of view of logical correctness, the whole step of the clock $C0$ is executed together with the substeps in the first thread, but operationally, accordingly to the previous example, it only makes sense to execute it together with the first substep. The constructivity should not change for this example, since the only difference is the visibility of the clocks.

The parallel examples considered so far only contain one new clock declaration, but the following examples shown in Figure 3.11 uses two unrelated clocks defined in different threads. Thereby, the first two examples define the two unrelated clocks $C1$ and $C2$, whereas the example on the right-hand side only defines $C1$. Since this third example is equivalent to the one discussed above in Figure 3.10 (c), the execution should not change to this one. Furthermore, the example in the middle does also not really change the program, but only adds the clock $C2$. So again for this example, the constructive execution should not change, since it should not depend only on the visibility of a clock. Finally, the example on the left-hand side of the figure really defines unrelated substeps having to be executed. Based on the discussion, the actions $\mathcal{A}_1, \mathcal{A}_3$ are executed together even if the substeps belong to different clocks. After that, the substeps can be executed independently due to the unrelated substeps. The following dependencies are possible:



Hence, it looks curious that the actions *right after* a `pause` statement of clock $C0$ are executed synchronously, even if they logically belong to unrelated substeps. But according to the discussion above, it seems to be the right definition since it does not depend on the visibility of clocks. Furthermore, this definition will match very well with the definition of the semantics and the compilation as it is presented in this thesis. But keep in mind that is is only one possible definition of constructivity that is taken here. One could imagine different ways to this end.

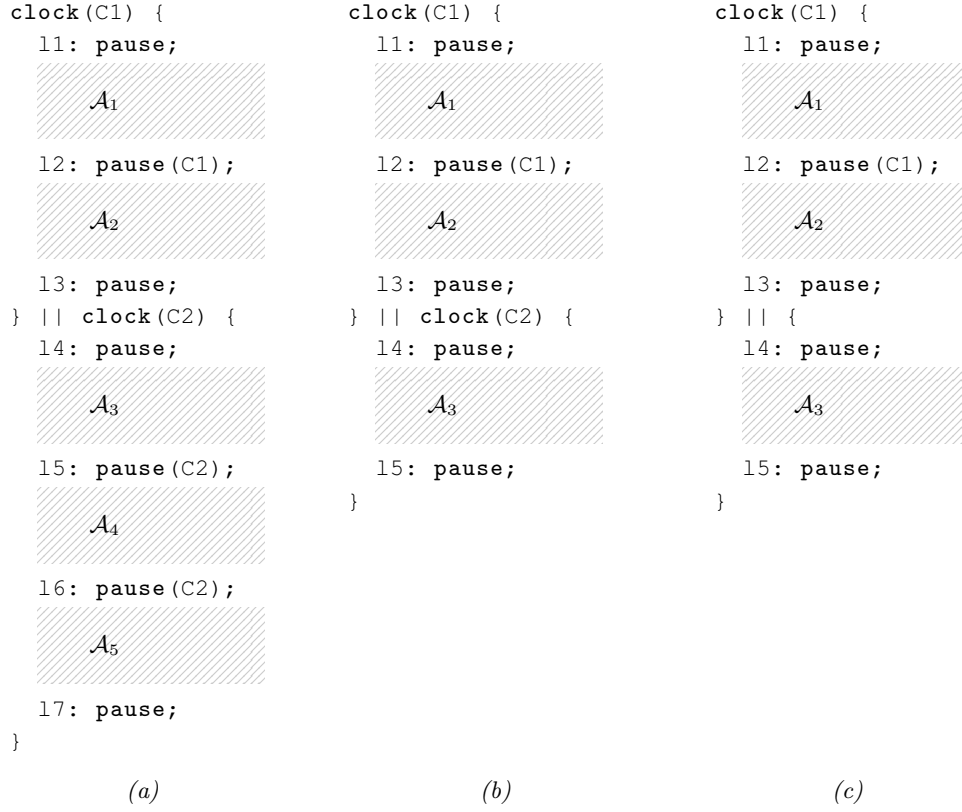


Fig. 3.11. Examples: Parallel Execution with Unrelated Clocks

3.3.3 Steps and Instants

Despite of the synonymous usage of the terms *step* and *instant* in pure QUARTZ, only the term *step* (of a clock) has been used in the context of the extension so far in this chapter, but not the term *instant*. However, both will get now a concrete meaning for the extension to be used in the following.

Like in pure QUARTZ, a step (of a clock) ranges from one **pause** statement to the next one (of this clock). The extension allows one to divide such a step by substeps of a lower clock. Consider the illustration in Figure 3.12 showing the steps of the clocks drawn on the right-hand side. The step of clock C0 ranges from label 11 to label 13, and also from label 14 to label 15 in the second thread. A step of clock C1 ranges e. g. from label 12 to label 13. According to the discussion above, the actions \mathcal{A}_1 and \mathcal{A}_3 are executed together, i. e. means in the same *instant*. Hence, the actions can influence each other (based on the defined constructivity), but actions executed in different instants can influence each other only if the dependencies follow the execution order. Instants can be seen as the *smallest* execution steps according to the constructive execution. Hence, the boxes summarizing the abstract action sets in the dependency diagrams above represent the instants of the program execution.

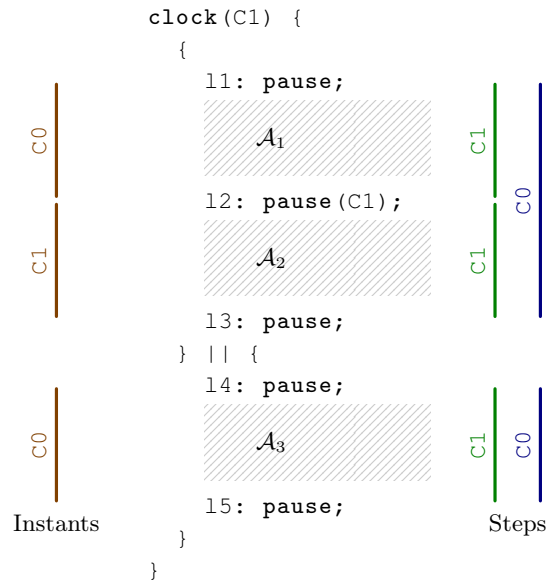


Fig. 3.12. Comparison of the Steps and Instants

The terms *step* and *instant* can be summarized as follows: a step ranges from a `pause` statement of a clock to the next `pause` statement of the *same* clock, whereas an instant ranges from a `pause` statement of a clock to the next `pause` statement of *any* clock. Remark, that the terminology generalizes the one of pure QUARTZ, because, since there only exist `pause` statements of the module clock, each step coincides with an instant. Finally, a second remark is given related to ESTEREL. In [Berr97b], Berry described an implicit signal called `tick` representing the activation clock (module clock) which holds in each *instant* and he introduced `pause` as an equivalent to the statement `await tick`, which waits for the next occurrence of the `tick` signal. In the context of the extension, this view extends to `pause` statements waiting of clock tick of their associated clock. If this clock occurs, the next instant of this clock is executed. However, the step of this clock is finally finished when the next `pause` statement of this clock is reached, and not necessarily after the instant itself. Hence, this interpretation fits also very well with the view introduced by Berry. Also recall the fact that the instants of clock `C0` are executed synchronously, even if they occur in parallel threads where unrelated clocks are defined. Interpreting the `pause(C0)` as an statement waiting for the next occurrence of the clock `C0` does also exactly lead to this above definition.

3.4 Summary

The clock refinement extension to QUARTZ has been informally introduced in this chapter based on an example and by discussions about the behavior of characteristic QUARTZ statements. The extension allows one to define clocks in addition to the already existing module clock. Thereby, a macro step of pure QUARTZ can be split into substeps based on

the new clocks. The substeps behave like macro steps on their own clock level and variables defined for them can change their value with each substep, and therefore, multiple times during a macro step of the module. The new clocks can be arbitrarily nested, and therefore, arbitrary many abstraction layers based on the clocks can be introduced, where each one follows the synchronous paradigm. However, the synchronous abstraction introduces the demand for constructive execution, which was also considered in this chapter by examples. It was explained which dependencies are allowed between substeps based on lower clocks, and how the (constructive) execution order is assumed. The following chapter will formally define the semantics of the extension, and together with that, it also defines a constructive execution.

Chapter 4

Formal Semantics

The semantics of pure QUARTZ is formally defined in [Schn09] by means of SOS rules as already explained in Section 2.2.4. Thereby, the rules define the behavior of each statement and are then used in an interpreter to define the complete (constructive) execution of a program. In this chapter, the rules and also the interpreter are reformulated to define the semantics of the extension, and enhances a first trial of formalizing the semantics presented in [GeBS10a] by handling local declarations and also by using a more convenient representation of the notion of instants as presented in the previous chapter.

As for pure QUARTZ, the SOS rules are also separated into *transition rules* and *reaction rules*. Thereby, the reaction rules determine the values of the variables for an instant, and the transition rules perform the execution of the instant on the program. In contrast to pure QUARTZ, a step of the extension can consist of multiple instants, hence, the execution of a step is based on multiple applications of transition and reaction rules. Furthermore, a variable can get its value during substeps of its clock, hence, it is not *known* from the beginning of the step. This is also not possible for pure QUARTZ, since the variables are all determined in each instant. However, the rules for the extension have to deal with unknown values to proceed the execution.

Each instant of the extension is based on a certain clock that is chosen before the rules are applied. Due to data dependencies between unrelated clocks, one choice of a clock could lead to an invalid execution whereas another choice would lead to a correct execution. Choosing clocks can be considered as *scheduling* of the execution and is discussed at the end of the chapter. The result are restrictions for the definition of constructivity which will ensure *scheduling independent* executions.

This chapter is structured as follows: first, some general definitions used by the semantics are given, which are then followed by the general introduction of transition and reaction rules. Both sets of rules are defined and explained in detail. Afterwards, the rules are used to define the complete behavior of the extension by an interpreter. The chapter concludes with a summary.

4.1 Definitions

The definition of the semantics is based on some formal notations to express e. g. the relation of clocks, or the value of a variable. Similar definitions can be found in [Schn09], but some

clock-related additions are given here. The following definitions depend on the program, but since in this thesis only one program is considered at once, the relation is clear from the context, and the program is not explicitly mentioned. Variables are used in programs to store values and they are covered by the following definition.

Definition 1 (Variables). *A program has a disjoint set of input variables \mathcal{V}^{in} , of output variables \mathcal{V}^{out} , and of local variables \mathcal{V}^{loc} . The set of all variables in a program is defined as (where $\dot{\cup}$ is the disjoint union of sets):*

$$\mathcal{V} := \mathcal{V}^{\text{in}} \dot{\cup} \mathcal{V}^{\text{out}} \dot{\cup} \mathcal{V}^{\text{loc}}$$

Furthermore, the sets of memorized variables \mathcal{V}^{mem} , and event variables \mathcal{V}^{eve} provide a second classification of all variables.

Thereby, each locally declared variable, even if the declaration is nested in loops, is contained in \mathcal{V}^{loc} . In programs of the extension, variables receive values during a step of their clock and they keep the value until the end of the step. However, as long as no value is assigned, the variable is considered as *unknown*, and if two conflicting values are assigned, the variable is considered as *invalid*. The following definition takes care of the values a variable can hold.

Definition 2 (Domain & Extended Domain). *Depending on the declaration, each variable x has an associated domain $\text{dom}(x)$ containing all values the variable can be assigned with. The extended domain of a variable x is defined by $\text{dom}_{\perp, \top}(x) := \text{dom}(x) \cup \{\perp, \top\}$. Thereby, the interpretation of the value \perp is unknown and the one of \top is invalid. Finally, the default value of a variable x is denoted by $\text{default}(x)$.*

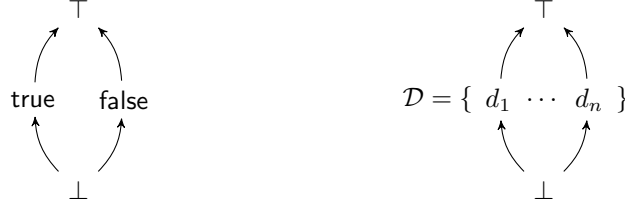
A variable x of Boolean type has e.g. the $\text{dom}(x) = \{\text{true}, \text{false}\}$ and the default value $\text{default}(x) = \text{false}$. Other domains are not explicitly mentioned in this thesis. Environments are used to assign a value to each variable. However, due to schizophrenia, a local variable can occur more than once in an instant, hence, all values must be handled by the environment. Therefore, an environment does not assign a single value to a variable, but a list of values. Since even for incarnated variables, only one of them is visible at a time, an additional *pointer* function stores an index for each variable identifying the value in the list, which has to be used.

Definition 3 (Environment). *An environment \mathcal{E} maps each variable $x \in \mathcal{V}$ to a list of values of its extended domain $\text{dom}_{\perp, \top}(x)$, i. e. $\mathcal{E}(x) = [v_0, \dots, v_n]$, with $v_i \in \text{dom}_{\perp, \top}(x)$. The function \mathfrak{h} represents a pointer mapping each variable to an index to access a value of the list. The access of the value is defined by $\mathcal{E}^{\mathfrak{h}}(x) = v_i$, with $\mathfrak{h}(x) = i$. An evaluation of an expression is defined by $\llbracket \tau \rrbracket_{\mathcal{E}}^{\mathfrak{h}}$, where for each variable the value identified by the function \mathfrak{h} is used. An update of the function \mathfrak{h} for a variable x is defined by:*

$$[\mathfrak{h}]_x^n(y) := \begin{cases} n & , \text{if } x = y \\ \mathfrak{h}(y) & , \text{else} \end{cases}$$

Hence, $[\mathfrak{h}]_x^n$ changes the index assigned to variable x to n , and keeps the indices for all other variables.

Definition 4 (Lattice of Extended Domain). A lattice of the values of an extended domain is given by the order \triangleleft which is defined as (for Booleans and for arbitrary domains):



The lattice is also extended to lists of values in the following way:

$$[v_0, \dots, v_n] \triangleleft [w_0, \dots, w_k] \Leftrightarrow \begin{cases} v_i \triangleleft w_i, 0 \leq i \leq n & , \text{if } n \leq k \\ v_i \triangleleft w_i, 0 \leq i \leq k \text{ and } v_i = \perp, k < i \leq n & , \text{else} \end{cases}$$

Hence, the lists are virtually extended with \perp to the same length.

Definition 5 (Partial Order, Union and Updates of Environments). A partial order of environments is given by extending the order of value list to environments. Hence, an environment \mathcal{E}_1 is smaller than environment \mathcal{E}_2 (greater resp.), iff the following holds:

$$\mathcal{E}_1 \sqsubseteq \mathcal{E}_2 \Leftrightarrow \forall x \in \mathcal{V}. \mathcal{E}_1(x) \triangleleft \mathcal{E}_2(x)$$

The union of environments is used to combine the values of environments into a single one, thereby it is defined for two environments \mathcal{E}_1 and \mathcal{E}_2 by the supremum of the values stored in the environments:

$$(\mathcal{E}_1 \dot{\sqcup} \mathcal{E}_2)(x) := \mathcal{E}_1(x) \dot{\sqcup} \mathcal{E}_2(x)$$

An update of an environment for a variable is defined by:

$$[\mathcal{E}]_{(x,n)}^v(y) := \mathcal{E} \dot{\sqcup} \{(x, [\perp^{n-1}, v])\}$$

Hence, the value of x at the position n of the value list is updated according to the lattice of its domain with the value v . All other values are kept.

Especially the last definition for updating an environment is important, since it does not only change the value, but update it according to the lattice. In this way, the monotonicity of all operations performed for environments are kept. Practically, this means that if the value of a variable is \perp , it is changed to the new value, but if the variable already has a concrete value, it is changed to \top (if the both values are conflicting). In this way, write conflicts are covered.

Clocks have been introduced to define substeps of an already existing macro step and they are used to identify the instants which are executed.

Definition 6 (Clocks & Partial Order of Clocks). The set of clocks of a program is denoted by \mathcal{C} including the module clock $\mathcal{C}0$. The clock of a variable x is referred to by $\text{clock}(x)$. A partial order of clocks (\mathcal{C}, \preceq) is given by the clock tree and it can be directly derived from the program. Thereby, $c_1 \preceq c_2$, iff the clock c_1 is declared in the scope of clock c_2 . The relation symbols \succeq, \succ, \prec are used accordingly. Additionally, two clocks are said to be unrelated $c_1 \# c_2$ iff neither $c_1 \preceq c_2$ nor $c_1 \succeq c_2$ holds.

Definition 7 (Environment Restriction). A restriction of an environment \mathcal{E} with respect to $\odot c$ (where $\odot \in \{\succ, \succeq, \prec, \preceq, \neq, \neq_s, \neq_c, \neq_d\}$) is defined as follows:

$$(\mathcal{E})_{/c_{s\odot}} c(x) := \begin{cases} \mathcal{E}(x) & \text{if } \text{clock}(x) \odot c \\ [\perp] & \text{otherwise} \end{cases}$$

Thus, $(\mathcal{E})_{/c_{s\neq}} c$ describes the environment where all variables with a clock lower or equal to c are set to $[\perp]$, the values of all other variables in \mathcal{E} are kept.

4.2 Overview

As already explained in Section 2.2.4, the traditional style of *Structural Operational Semantics (SOS)* [Plot81, Moss06] cannot be directly used for (imperative) synchronous languages, because the actual execution does not completely follow the program structure. Therefore, the SOS rules are separated into two sets: The reaction rules determine an environment for the next instant. After such an environment is discovered, the transition rules follow the program structure based on this environment and transform the program for the next instant.

This idea is kept for the extension, and based on the constructivity discussed in Section 3.3, the semantics is defined based on the instants: applying the transition rules means executing one instant of a certain clock. Obviously, due to unrelated clocks, different choices for the clock can exist, and based on the program, this can be a deterministic or non-deterministic choice. However, even for non-deterministic choices, which means a really unrelated execution, at the end of the module step the same state will be reached because the whole model is still deterministic.

Parallel execution of unrelated clocks and also of only refined clocks needs to synchronize on each `pause` statement of a common clock. If one thread reaches a `pause` of a common clock before the other one does, it has to *wait*. To represent this behavior with the rules, an additional statement is introduced to model this *barrier*.

Additional Statements

The `pause` statement of QUARTZ can be interpreted in several ways, which do not differ in the single clock case, but in the context of refined clocks. In Section 3.3.3, a view to the `pause` statement has been developed which coincides with an interpretation originally introduced by Berry [Berr97b]. Thereby, in the single-clock case of ESTEREL, a dedicated clock signal `tick` is used which holds in every instant, and the `pause` statement is seen as a replacement for `await tick`, which only proceeds when the *next* trigger signal occurs, hence in the next instant. This view is emphasized in the following for the definition of the semantics of extended QUARTZ. Therefore, the `pause` statement is renamed to the statement `await clock`. Analogous to the original `await` statement, the transition rules also use an *immediate* variant of the statement, which is written `immediate await clock`. Finally, the following statements are used instead of the original ones:

$$\begin{aligned} \text{pause} & \quad \equiv \quad \text{await clock}(C_0) \\ \text{pause}(C) & \quad \equiv \quad \text{await clock}(C) \end{aligned}$$

Additionally, the following statement is introduced for the SOS rules:

`immediate await clock(C)`

Note that the statement `immediate await clock` has no counterpart in the original set of statements and thus it can also not be used in (real) source code. But especially this statement allows to formulate the rules in an easy and accessible way. Therefore, it is also reasonable to rename the statements here, to emphasize the relationship of `await clock` and its `immediate` counterpart.

4.3 Transition Rules

Transition rules define how the program is transformed during the execution of an instant resulting to a new program which is considered in the following instant. The rules are defined on the program structure for each statement, hence, the whole program is considered as a single statement. For an instant, the values of the variables must be given by an environment, but, in contrast to pure QUARTZ, the environment does not need to be defined for all variables, since either variables of unrelated clocks do not affect the execution or variables of a higher clock can be set in a following instant. However, the variables needed for the execution of the instant must be set, and therefore, the value \perp does not appear in the transition rules. If e.g. a condition would be evaluated to \perp with the given environment, the transition rules cannot be applied and the execution fails. This section presents the general form of the transition rules first, followed by the rules for each statement.

4.3.1 General Form of the Rules

The transition rules for the extension are of the following form:

$$\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}^{\text{next}}, C \rangle$$

A rule of this form specifies that the statement \mathcal{S} is transformed in an instant of clock c with the given variable environment \mathcal{E} to the statement \mathcal{S}' . Remember that the transition rules are only applied for a single instant which is possibly just a part of a higher step and that they are applied according to the Abstract Syntax Tree (AST). The individual symbols are described in the following.

- *Instant Clock c*
The currently processed instant is of clock c , and furthermore, the very first instant which is executed for a program when it is started is of clock C0. Later it is required that at least one `pause` statement of the clock has been reached in the instant before, and c must be *one* of the *smallest* reached clocks. Note that there can be only a choice between unrelated clocks.
- *Environment \mathcal{E}*
The environment \mathcal{E} is used for the execution of the current instant. Thereby, \mathcal{E} can be only partially defined due to variables related to higher clocks which are possibly set in a later instant but still within a step of their clock, and also due to variables which belong to clocks which are unrelated to the instant clock. However, all variables which are *needed* for this step have to be defined by the environment \mathcal{E} .

- *Incarnation Levels \bar{h} and h'*
The incarnation level function maps each variable to its incarnation level, i. e. the number of scopes of the variable that have been entered in this instant so far. Thereby, \bar{h} is the incarnation level *before* the statement, i. e. it counts all scopes that have been entered before reaching this statement, and h' is the updated incarnation level *after* the statement, i. e. it is updated by the scopes additionally entered during processing \mathcal{S} .
- *Statement Clock $C_{\mathcal{S}}$*
The statement clock is the smallest visible clock at the current position in the AST. The clock is updated whenever a clock declaration is executed, and it is required as a context for the considered statement, e. g. the abortion statement can abort the execution of clocks defined outside, but not of clocks defined inside, thus this parameter allows to separate them. The execution of the program is started with C_0 for this parameter, since the module clock is the only clock defined for the whole program.
- *Origin Statement \mathcal{S} & Residual Statement \mathcal{S}'*
The rule is applied to the statement \mathcal{S} to process an instant and it will produce a residual statement \mathcal{S}' being processed in the following instant.
- *Assignments $\mathcal{A}^{\text{next}}$*
Like the original transition rules, the delayed assignments executed in the instant are collected in this set. They are used to determine the values which have to be carried over to the following steps.
- *Reached Clocks \mathcal{C}*
The transition rules for pure QUARTZ contain a simple flag to indicate whether a **pause** statement to complete the instant has been reached or not during the execution. For the extension, it has to be determined which clock the next instant should be of. Therefore, the clocks of the **pause** statements reached in the instant are contained in the set \mathcal{C} . The relation between this original flag t and this set is: $t = \text{true} \Leftrightarrow \mathcal{C} = \{\}$.

The transition rules of pure QUARTZ have the invariant that if no **pause** statement is reached during the execution (flag t is **true**), the residual statement is **nothing**. In this case, the statement is completely executed and nothing remains to be done for the next instant. The same invariant also holds for the extension: $\mathcal{C} = \{\}$ implies that the residual statement is **nothing**. The rules are given and explained in the following.

4.3.2 Basic Statements

The definition of the transition rules for some basic statements are shown in the Figures 4.1 and 4.2, they are explained in the following.

- *Rules for Assignments*
There are two different kinds of assignments: immediate assignments and delayed assignments. In the rules of pure QUARTZ, the preconditions for the assignments are not needed, since there the transition rules are only applied with a complete environment. In the extension, not all variables must be assigned at the beginning of a step, and therefore, a variable could be unknown. To ensure that all expressions during execution of the instant are evaluated to real values and not to \perp or \top , the preconditions are given here. This also allows to formulate the simulator in a simpler way, since these conditions do

Assignments

$$(a_1) \quad \frac{\llbracket \tau \rrbracket_{\mathcal{E}}^h = \mathcal{E}^h(x) \wedge \mathcal{E}^h(x) \notin \{\perp, \top\}}{\langle \mathcal{E}, \bar{h}, C_S, x = \tau \rangle \xrightarrow{c} \langle \bar{h}, \text{nothing}, \{\}, \{\} \rangle}$$

$$(a_2) \quad \frac{\llbracket \tau \rrbracket_{\mathcal{E}}^h \notin \{\perp, \top\}}{\langle \mathcal{E}, \bar{h}, C_S, \text{next}(x) = \tau \rangle \xrightarrow{c} \langle \bar{h}, \text{nothing}, \{\text{next}(x) = \tau, \bar{h}\}, \{\} \rangle}$$

Time Consuming Statements

$$(p_1) \quad \langle \mathcal{E}, \bar{h}, C_S, \text{await clock}(C) \rangle \xrightarrow{c} \left\langle \bar{h}, \left\{ \begin{array}{l} \text{immediate} \\ \text{await clock}(C) \end{array} \right\}, \{\}, \{C\} \right\rangle$$

$$(p_2) \quad \frac{c = C}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{immediate} \\ \text{await clock}(C) \end{array} \right\} \right\rangle \xrightarrow{c} \langle \bar{h}, \text{nothing}, \{\}, \{\} \rangle}$$

$$(p_3) \quad \frac{c \neq C}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{immediate} \\ \text{await clock}(C) \end{array} \right\} \right\rangle \xrightarrow{c} \left\langle \bar{h}, \left\{ \begin{array}{l} \text{immediate} \\ \text{await clock}(C) \end{array} \right\}, \{\}, \{C\} \right\rangle}$$

Clock Definitions

$$(c_1) \quad \frac{C \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C, S \rangle \xrightarrow{c} \langle \bar{h}', S', A, C \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{clock}(C) \{ S \} \rangle \xrightarrow{c} \langle \bar{h}', \text{clock}(C) \{ S' \}, A, C \rangle}$$

$$(c_2) \quad \frac{C = \{\} \wedge \langle \mathcal{E}, \bar{h}, C, S \rangle \xrightarrow{c} \langle \bar{h}', S', A, C \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{clock}(C) \{ S \} \rangle \xrightarrow{c} \langle \bar{h}', S', A, C \rangle}$$

Conditional Statements

$$(i_1) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^h = \text{true} \wedge \langle \mathcal{E}, \bar{h}, C_S, S_1 \rangle \xrightarrow{c} \langle \bar{h}', S'_1, A_1, C_1 \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{if}(\sigma) S_1 \text{ else } S_2 \rangle \xrightarrow{c} \langle \bar{h}', S'_1, A_1, C_1 \rangle}$$

$$(i_2) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^h = \text{false} \wedge \langle \mathcal{E}, \bar{h}, C_S, S_2 \rangle \xrightarrow{c} \langle \bar{h}', S'_2, A_2, C_2 \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{if}(\sigma) S_1 \text{ else } S_2 \rangle \xrightarrow{c} \langle \bar{h}', S'_2, A_2, C_2 \rangle}$$

Fig. 4.1. Transition Rules I (Basic Statements I)

Sequence

$$(s_1) \quad \frac{C_1 \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}'_1, \mathcal{A}_1, \mathcal{C}_1 \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1; \mathcal{S}_2 \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}'_1; \mathcal{S}_2, \mathcal{A}_1, \mathcal{C}_1 \rangle}$$

$$(s_2) \quad \frac{C_1 = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \xrightarrow{c} \langle \bar{h}_1, \mathcal{S}'_1, \mathcal{A}_1, \mathcal{C}_1 \rangle \wedge \langle \mathcal{E}, \bar{h}_1, C_S, \mathcal{S}_2 \rangle \xrightarrow{c} \langle \bar{h}_2, \mathcal{S}'_2, \mathcal{A}_2, \mathcal{C}_2 \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1; \mathcal{S}_2 \rangle \xrightarrow{c} \langle \bar{h}_2, \mathcal{S}'_2, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_2 \rangle}$$

Loop

$$(l) \quad \frac{C \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{do } \mathcal{S} \text{ while } (\sigma); \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}'; \text{if } (\sigma) \text{ do } \mathcal{S} \text{ while } (\sigma), \mathcal{A}, \mathcal{C} \rangle}$$

Parallel Threads

$$(p) \quad \frac{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \xrightarrow{c} \langle \bar{h}_1, \mathcal{S}'_1, \mathcal{A}_1, \mathcal{C}_1 \rangle \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_2 \rangle \xrightarrow{c} \langle \bar{h}_2, \mathcal{S}'_2, \mathcal{A}_2, \mathcal{C}_2 \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \parallel \mathcal{S}_2 \rangle \xrightarrow{c} \langle \text{Max}(\bar{h}_1, \bar{h}_2), \mathcal{S}'_1 \parallel \mathcal{S}'_2, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \rangle}$$

Local Declaration

$$(d) \quad \frac{\langle \mathcal{E}, [\bar{h}]_x^{\bar{h}(x)+1}, C, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \{\alpha x; \mathcal{S}\} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}$$

Nothing

$$(n) \quad \langle \mathcal{E}, \bar{h}, C_S, \text{nothing} \rangle \xrightarrow{c} \langle \bar{h}, \text{nothing}, \{\}, \{\} \rangle$$

Fig. 4.2. Transition Rules II (Basic Statements II)

not have to be checked there. Both kinds of assignments are derived by the rules to the residual statement **nothing**, since they are completely executed in this instant, and no **pause** statement is reached. Rule (a_1) handles the immediate assignment which is, if the preconditions are fulfilled, just consumed. The delayed assignments are handled by Rule (a_2), which collects the assignment together with the incarnation function to decide later which incarnation of x should be assigned.

- *Rules for Time Consuming Statements*

As already explained, **pause** is renamed to **await clock** for the SOS rules, and therefore, the rule (p_1) handles the ordinary **pause**. As in pure QUARTZ, the executed instant ends at each **pause** statement that is reached, but for the extension the clock of the **pause** is also collected in the set \mathcal{C} . Additionally, the **pause** is not just transformed to **nothing**, but the

residual statement is `immediate await clock`. The other both rules (p_2) and (p_3) handle the `immediate await clock` statement with different preconditions. This statement can be consumed when an instant of the related clock is processed, i. e. $c = C$ holds. Otherwise, if the clocks do not match, the instant ends and it behaves like the ordinary `await clock`. In this way, as long as no instant of its clock is processed, the `immediate await clock` behaves like a barrier. In the context of pure QUARTZ, each instant is executed with the module clock, and in this case the `immediate await clock(C0)` is equivalent to `nothing`.

- *Rules for Local Clock Definitions*

When the scope of a local clock declaration is entered, the rules update the *statement clock* for the substatement. The rules (c_1) and (c_2) distinguish the cases where the local block is completely executed or not, in the latter case, the whole block can be omitted for the residual statement.

- *Rules for Conditional Statements*

The conditional `if` statement just selects one of its branches depending on the evaluation of the condition. The evaluation of the expression must lead to either `true` or `false`, because otherwise, either the evaluation failed, or not enough variables are known in the environment.

- *Rules for Sequence Statement*

The behavior of the sequence of two statements is expressed by the rules (s_1) and (s_2) in Figure 4.2. Thereby, the first statement is executed in the context of the whole sequence and the second one is only executed when the first one terminates in the instant. Therefore, the distinction can be made by the set \mathcal{C}_1 , which collects the clocks for which a `pause` statement is reached in \mathcal{S}_1 . If this set is empty, the whole statement \mathcal{S}_1 is consumed, and the second statement \mathcal{S}_2 is executed in the same instant. Otherwise, the instant reaches a `pause` inside \mathcal{S}_1 (which is derived to \mathcal{S}'_1), and the residual statement is the sequence $\mathcal{S}'_1; \mathcal{S}_2$.

- *Rules for Loop*

For the `do . . . while (σ)` loop, the body is simply executed, and when the end is reached, it is checked whether it should be executed again or not. As the residual statement shows in Rule (l), the body is executed and after that an `if` condition is added to check the condition for a possible restart. Note also, that in synchronous languages instantaneous loops are not allowed and the instant must end inside the loop body, as additionally required by the precondition of the rule. However, this rule is also of interest for local variables, because it possibly duplicates declarations. Note that the declarations are put into a sequence and therefore, the execution of an instant can enter multiple scopes in one instant (or step), but it cannot be inside more than one declaration at a time.

- *Rules for Parallel Statement*

One of the interesting statements for the extension is the parallel statement, since it allows the declaration of unrelated clocks for independent execution. Nevertheless, the transition rule for the parallel statement is simple, since it applies the context of the instant to both threads, executes the instants simultaneously and combines the results. The residual statement is the parallel combination of both residual statements of the threads. However, one detail is missing here, because if both threads are completely executed, the whole

parallel should also be completely executed and the residual statement should be just `nothing`, i. e. implicitly `nothing || nothing` has to be treated as `nothing`. Finally, the incarnation level of both threads is combined by $\text{Max}(h_1, h_2)$ taking the maximal value for each variable: a local variable can be either defined in the one or the other thread, for each variable the incarnation level can be only increased by one thread.

- *Rule for Local Declaration*

Rule (d) handles the local declarations by updating the incarnation level for the defined variable to execute the substatement. Finally, the whole statement is derived to the residual statement of the substatement. The declaration itself is removed, since the scope is entered in this instant, hence the following instants are automatically executed *inside* this scope.

- *Rule for Nothing*

Finally, Rule (n) handles the statement `nothing` straightforwardly, since the statement has simply no real behavior, but it is given here for completeness.

On the one hand, the introduction of the new statement `await clock` and especially `immediate await clock` looks curious, but it can now be justified with the help of the parallel statement. In the rules for pure QUARTZ, a `pause` statement is consumed in the instant it is *reached* and the following instant will start from this position. The rules for the extension collect the clocks of the `pause` statements which are reached and depending on those, the clock of the next instant can be determined. Due to the expected behavior, parallel threads need to synchronize on `pause` statements of the same clock. When `pause` statements would be completely consumed, the information that one thread needs to wait for a higher clock is lost. A second solution could be to keep the `pause` of the higher clock, but this decision cannot be taken locally because it depends on the `pause` statements reached in the other thread (and vice versa). Therefore, the introduction of the statement `immediate await clock` is an elegant way to define a *barrier* which can only be got over by an instant of the right clock. Informally, the threads are *marked* with the clocks they need to process. Defining the rules for the statements in this way has the *side-effect* that also unrelated clocks are handled properly. This would require more effort otherwise.

To illustrate this, consider the code examples in Figure 4.3. The example only focuses on the correct execution of the substeps, and data dependencies are not taken into account. The original program statement is given by \mathcal{S}_0 in Figure 4.3 (a) consisting of parallel threads where a refined clock `C1` is declared in the first one. The whole program starts with an instant of clock `C0`. Both threads should synchronize on the `pause` statements with labels `l2` and `l3`. Hence, the actions \mathcal{A}_1 and \mathcal{A}_3 are executed together in an instant of clock `C0`, the actions \mathcal{A}_2 are then executed in an instant of clock `C1`, and finally the actions \mathcal{A}_4 are executed in an instant of clock `C0`. Hence, in the first instant, the following transition is made based on a previously determined environment:

$$\langle \mathcal{E}_0, \bar{h}, C0, \mathcal{S}_0 \rangle \xrightarrow{C0} \langle \bar{h}', \mathcal{S}_1, \mathcal{A}_1 \cup \mathcal{A}_3, \{C0, C1\} \rangle$$

The transition also introduces the `immediate await clock` statements and executes the actions \mathcal{A}_1 and \mathcal{A}_3 . Depending on the collected clocks, the second instant is of clock `C1` and is completed by the transition:

$$\langle \mathcal{E}_1, \bar{h}, C0, \mathcal{S}_1 \rangle \xrightarrow{C1} \langle \bar{h}', \mathcal{S}_2, \mathcal{A}_2, \{C0\} \rangle$$

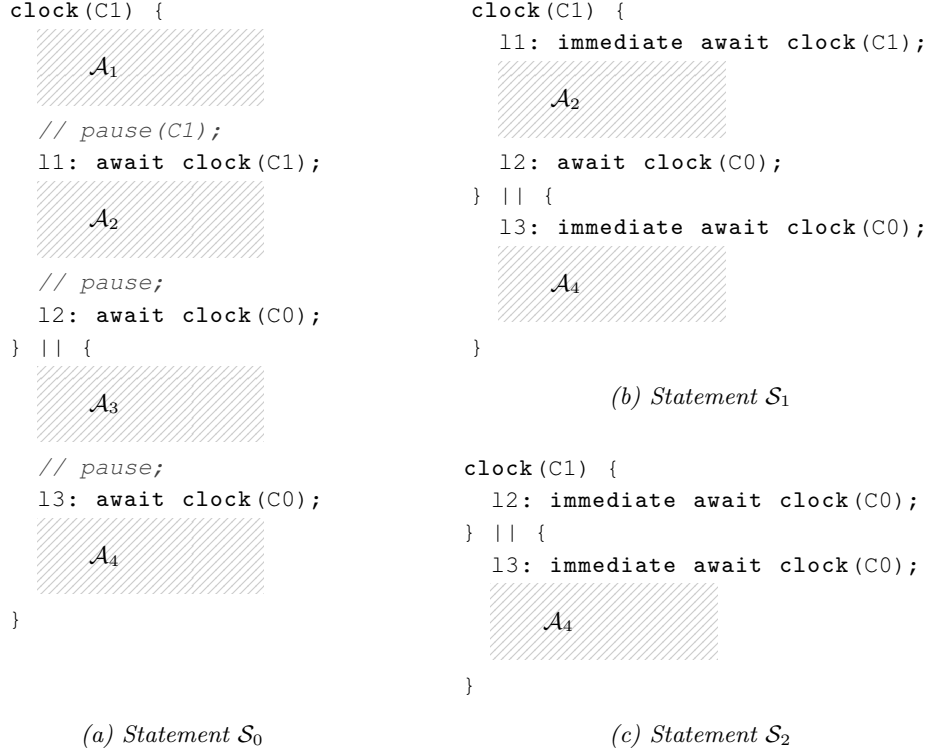


Fig. 4.3. Example: Transition Rules for `pause` Statements

This transition executes the actions \mathcal{A}_2 , because `immediate await clock(C1)` can be crossed in the first thread, but the barrier for clock C0 is not crossed in the second one. Hence, the importance of the clock barrier introduced by the `immediate await clock` statement for clock C0 can be seen. Without this barrier, the decision whether also the actions \mathcal{A}_4 in the second thread has to be executed or not cannot be taken locally. Again, this additional statement allows one to encode the formerly reached `pause` statements in the residual statements turning out to be an elegant way for defining the transition rules. In pure QUARTZ, this could also be introduced but it would not make a difference because there only the clock C0 exist which holds in each instant: the barriers would be crossed every time.

4.3.3 Strong Preemption

The preemption statements, `abort` and `suspend`, influence the execution of their substatements in each step. Thereby, two versions are distinguished for the (strong) preemption identified by the keyword `immediate`. The delayed preemption (without `immediate`) does not affect the execution in the step the statement is entered, but only when the control flow rests inside a preemption block at a label which is associated with the statement's clock (or any higher clock). On the other hand, the immediate preemption can take place directly when the statement is started.

In pure QUARTZ, this behavior is reflected by the transition rules in the following way: the first instant is executed for the delayed preemption block and then it is changed to its immediate counterpart for the residual statement. In this way, no preemption can take place in the first step, but after the first step it can occur in every step. However, this solution does not directly apply to the extension, because the first step can consist of multiple instants (of a lower or unrelated clock). Therefore, the residual statement cannot be changed until all instances of the first step are executed. The rules are explained in the following.

Strong Abortion

The transition rules for the strong abortion statements are shown in Figure 4.4. The main difference between the immediate and the delayed abortion is the different treatment of the first step.

(Strong) Abortion

$$\begin{array}{l}
(a_1) \quad \frac{\mathcal{C} = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle} \\
(a_2) \quad \frac{(\exists c \in \mathcal{C}. c \prec C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{abort } \mathcal{S}' \text{ when } (\sigma), \mathcal{A}, \mathcal{C} \rangle} \\
(a_3) \quad \frac{(\forall c \in \mathcal{C}. c \succeq C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \left\langle \bar{h}', \left\{ \begin{array}{l} \text{immediate await clock } (\text{Min}(\mathcal{C})) ; \\ \text{immediate abort } \mathcal{S}' \text{ when } (\sigma) \end{array} \right\}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(a_4) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true}}{\langle \mathcal{E}, \bar{h}, C_S, \text{immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \{\}, \{\} \rangle} \\
(a_5) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{immediate abort} \\ \mathcal{S} \\ \text{when } (\sigma) \end{array} \right\} \right\rangle \xrightarrow{c} \left\langle \bar{h}', \left\{ \begin{array}{l} \text{immediate abort} \\ \mathcal{S}' \\ \text{when } (\sigma) \end{array} \right\}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(a_6) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \wedge \mathcal{C} = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle}
\end{array}$$

Fig. 4.4. Transition Rules III (Strong Abortion)

First, the delayed abortion is considered, which only has to execute the first step without considering the abort condition. Rule (a₁) covers the case that the substatement is completely

executed in this instant and no `pause` statement is reached inside. In this case, the whole statement is completely finished, and nothing remains for the residual statement. The precondition of Rule (a_2) considers the case that at least one `pause` statement with a clock lower than the statement's clock C_S is reached. In this case, there is at least one additional instant to be executed before the first step entering the abort statement is finished. The residual statement is again handled as a delayed abort statement. Rule (a_3) is the most interesting rule and considers the case that all clocks of the `pause` statements reached during this instant are equal or higher to the statement's clock C_S . In this case, the first step entering the abort statement is completely executed, and in the next step, the abortion can possibly take place. Practically, the residual statement could be changed to the immediate variant. However, despite the step is finished for this statement, it cannot be decided locally if there is another parallel thread having to execute another instant. For this reason, the additional `immediate await clock` ensures that the abortion can only take place in the next step. Thereby, $\text{Min}(\mathcal{C})$ denotes the least clock contained in \mathcal{C} which (1) is not empty, and (2) contains only clocks higher or equal to C_S , hence, this minimum is uniquely defined. The execution of the statement can only proceed when the execution of its substatement can do so.

Second, the immediate version of the abort statement is considered. Since for this statement an abortion can occur, it checks for the condition whether to abort or not. If the condition is evaluated to `true` handled by the precondition of Rule (a_4) , the substatement is aborted, nothing is executed, and the residual statement is `nothing`. Rule (a_5) and Rule (a_6) handle the case that the condition is evaluated to `false`, the behavior is different, depending on, whether a `pause` statement is reached or not. If one is reached, the residual statement is just kept, otherwise the whole statement is executed and so also the abort statement is.

The difference for the immediate abort variant is that the condition can be checked in *each* instant. However, abortion will only take place in the first instant of a step since each variable occurring in the condition can only be of the statement's clock or higher, due to the visibility of clocks. Hence, they have their value for the whole step, and checking the condition again in each instant does not change the behavior.

To discuss the need for the additional `await immediate clock` in the residual statement for the delayed abortion in Rule (a_3) , consider the code example in Figure 4.5. The starting point is statement \mathcal{S}_0 being transformed after an instant is executed to \mathcal{S}_1 :

$$\langle \mathcal{E}_0, \bar{h}, C_0, \mathcal{S}_0 \rangle \xrightarrow{C_0} \langle \bar{h}', \mathcal{S}_1, \mathcal{A}_1, \{C_0, C_1\} \rangle$$

The first step of clock C_0 is finished in the first thread, but not in the second one, where an additional instant must be executed. The additional `await immediate clock` now ensures that abortion cannot occur for the abort statement because it is not entered until the next instant of clock C_0 is executed, hence, when the next step is started.

Strong Suspension

After having explained the rules for the abortion statement, the rules for the suspension can be basically assembled in the same way. The delayed suspension does not take place in the first step, but in each following one, and it is changed to the immediate variant after the first step. The transition rules are shown in Figure 4.6.

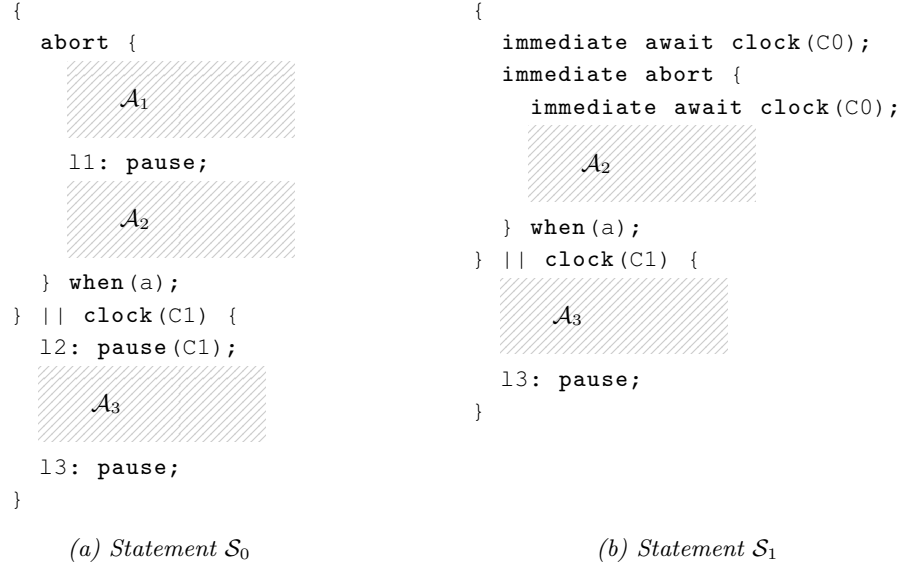


Fig. 4.5. Example: Strong Abortion Rules

The rules have the same preconditions as the rules for the abortion statement. Rule (s_1) covers the case that the substatement is completely executed in the instant, and no `pause` statement is reached inside. In this case, the whole statement is completely finished, and nothing remains for the residual statement. The precondition of Rule (s_2) considers the case that at least one `pause` statement with a clock lower than the statement clock C_S is reached. In this case, there is at least one additional instant to be executed before the first step entering the suspension statement is finished. Finally, Rule (s_3) considers the case that all clocks of the `pause` statements reached during this instant are equal or higher than the statement clock C_S . In this case, the first step entering the suspension statement is completely executed, and in the next step, the suspension can possibly take place. For the residual statement, the `immediate await clock` does not need to be added here, because the execution will not proceed here. Even if the condition would hold, the spirit of suspension is to execute nothing, and this is exactly what is expected.

In addition, the rules for the immediate version of the suspension statement have the same preconditions as the abortion rules. If the condition is evaluated to `true` for Rule (s_4), nothing is executed, and the residual statement is not changed, since the execution is just postponed and not aborted. If the condition is evaluated to `false`, the same two cases have to be distinguished: either a `pause` statement is reached inside (Rule (s_5)), or the whole block is consumed (Rule (s_6)) as well as the suspension statement.

4.3.4 Weak Preemption

In contrast to their *strong* counterparts, the *weak* preemption statements execute the current step when the preemption takes place before the step ends, the execution is either aborted or suspended. In pure QUARTZ, the execution is limited to the data-flow of the executed

(Strong) Suspension

$$\begin{array}{l}
(s_1) \quad \frac{\mathcal{C} = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{suspend } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle} \\
(s_2) \quad \frac{(\exists c \in \mathcal{C}. c \prec C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{suspend } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{suspend } \mathcal{S}' \text{ when } (\sigma), \mathcal{A}, \mathcal{C} \rangle} \\
(s_3) \quad \frac{(\forall c \in \mathcal{C}. c \succeq C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{suspend } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \left\langle \bar{h}', \left\{ \begin{array}{l} \text{immediate} \\ \text{suspend } \mathcal{S}' \text{ when } (\sigma) \end{array} \right\}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(s_4) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true}}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{immediate} \\ \text{suspend } \mathcal{S} \text{ when } (\sigma) \end{array} \right\} \right\rangle \xrightarrow{c} \left\langle \bar{h}', \left\{ \begin{array}{l} \text{immediate} \\ \text{suspend } \mathcal{S} \text{ when } (\sigma) \end{array} \right\}, \{\}, \{C_S\} \right\rangle} \\
(s_5) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{immediate} \\ \text{suspend } \mathcal{S} \text{ when } (\sigma) \end{array} \right\} \right\rangle \xrightarrow{c} \left\langle \bar{h}', \left\{ \begin{array}{l} \text{immediate} \\ \text{suspend } \mathcal{S}' \text{ when } (\sigma) \end{array} \right\}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(s_6) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \wedge \mathcal{C} = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{immediate suspend } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle}
\end{array}$$

Fig. 4.6. Transition Rules IV (Strong Suspension)

step, whereas with refined clocks, the execution can contain steps related to a lower clock. Therefore, in pure QUARTZ, the structure of the transition rules perfectly matches this execution because they consider one step at each time. For refined clocks, the transition rules only consider the instants, and a step related to a certain clock can contain multiple instants. Hence, some instants have to be processed until the real preemption takes place. However, note that the preemption condition (logically) holds for the whole step. For the abortion, the rules can be formulated in a straightforward way, because the abortion is simply taken at the end of a step. For the suspension, the rules are more tricky, because the original control-flow position has to be kept, since when the suspension takes place, the whole step is executed, but the following step will start at the same control-flow position as of the previous step. Finally, delayed preemption can be handled in the same way.

Weak Abortion

The transition rules for the weak abortion statements are shown in Figure 4.7. The rules for the weak abortion are similar to the ones for the strong abortion: after the first step of the statement clock, the keyword `immediate` is added. This is handled by the rules (a_1) ,

(Weak) Abortion

$$\begin{array}{l}
(a_1) \quad \frac{\mathcal{C} = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle} \\
(a_2) \quad \frac{(\exists c \in \mathcal{C}. c < C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{weak abort } \mathcal{S}' \text{ when } (\sigma), \mathcal{A}, \mathcal{C} \rangle} \\
(a_3) \quad \frac{(\forall c \in \mathcal{C}. c \succeq C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{weak immediate abort } \mathcal{S}' \text{ when } (\sigma), \mathcal{A}, \mathcal{C} \rangle} \\
(a_4) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true} \wedge (\forall c \in \mathcal{C}. c \succeq C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \{\} \rangle} \\
(a_5) \quad \frac{(\exists c \in \mathcal{C}. c < C_S \vee \llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false}) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{weak immediate} \\ \text{abort} \\ \mathcal{S} \\ \text{when } (\sigma) \end{array} \right\} \right\rangle \xrightarrow{c} \left\langle \bar{h}', \left\{ \begin{array}{l} \text{weak immediate} \\ \text{abort} \\ \mathcal{S}' \\ \text{when } (\sigma) \end{array} \right\}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(a_6) \quad \frac{\mathcal{C} = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle}
\end{array}$$

Fig. 4.7. Transition Rules V (Weak Abortion)

(a₂) and (a₃). The remaining rules take care of *real* abortion. Therefore, also three cases are distinguished which are not the same as for the strong variant.

In pure QUARTZ, the SOS rules for weak and strong abortion differ in the way the data flow, i. e. the executed assignments, are handled, but both versions abort the control flow. The difference for refined clocks is that for the weak abortion, not only the data flow is kept but also the control flow related to lower clocks, i. e. each instant which belongs to a smaller clock than the statement clock of the abort statement. Hence, in a similar way as the delayed abortion is extended with the **immediate** keyword at the *right* instant, the SOS rules for the weak abortion also needs to stop the execution at the *right* instant. The following cases are distinguished by the rules:

- The weak abort statement aborts the execution in the last instant which belongs to the step of the statement's clock. Rule (a₄) identifies the last instant by the **pause** statements which are reached: If all clocks are higher or equal to the clock, no smaller instants are executed for this step. If also the abort condition holds in this instant, the execution is aborted. Note that similar to pure QUARTZ, the actions \mathcal{A} are also executed in this instant.

- The Rule (a_5) covers the case where the execution can simply proceed. This is the case if either the abort condition is evaluated to **false** or an instant where the abortion does not take place is executed. Note that in the second case, it is not needed to evaluate the abortion condition. For the weak version of the abortion, it is only needed to *know* the condition in the last instant of a step.
- Finally, Rule (a_6) covers the case where the body of the abortion statement is consumed. Furthermore, this case does not care about the abortion condition, since in both cases the actions \mathcal{A} would be executed, as it is defined by the rule.

To sum up, the rules abort the execution of a step related to its statement clock in the last instant of such a step. Thereby, the actions of such an instant are also executed. In this way, the whole data flow of the step and the control flow related to smaller clocks are executed. From the preconditions of the rules, it can be seen that the evaluation of the abort condition to true or false is only needed in the case where the precondition

$$(\forall c \in \mathcal{C}. c \succeq C_S) \wedge \mathcal{C} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}, \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle$$

holds. If in this case, the abortion condition is evaluated to **true**, the Rule (a_4) is used, and if it is evaluated to **false**, the Rule (a_5) is used. More important is that the condition expresses that only **pause** statements of a clock which are higher or equal to the current statement clock are reached. Hence, the last instant of a step of C_S is executed. Thus, the evaluation of the condition σ is only needed in this last instant. Since the expression σ can only be composed of variables with a clock higher or equal to C_S , from the logical point of view, this expression holds the whole step. However, from the view of causality, the variables used in the expression can be set within the step and they only need to be known in the last instant.

Weak Suspension

The definition of the SOS rules for weak suspension is more complex. Like the weak abortion, the weak suspension executes all substeps of the affected step, but afterwards the control-flow is reset. In pure QUARTZ, this is simply handled by executing the data flow and not touching the control flow. However, for refined clocks, the control flow related to smaller clocks needs to be executed. Therefore, the control flow of the lower clocks has to be executed and when the whole step is finished, the initial position needs to be *restored*. The transition rules for the weak suspension are shown in Figure 4.8. The first rules consider the delayed suspension statement which just execute the first step according to the statement's clock and then adds the **immediate** keyword to the statement.

A hypothetical definition of the rules for the **immediate** version of the weak suspension is discussed now. Basically, for a step of the statement's clock, there are two possibilities: (1) the suspension condition holds or (2) the suspension condition does not hold. Both possibilities can be easily expressed by rules. Therefore, if the suspension condition does not hold, the execution can proceed without any special treatment. However, if the suspension condition holds, one step should be executed and afterwards the same situation should be present than before. In this case, the statement can be substituted with:

(Weak) Suspension

$$\begin{array}{l}
(s_1) \quad \frac{C = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak suspend } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle} \\
(s_2) \quad \frac{(\exists c \in \mathcal{C}. c \prec C_S) \wedge C \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak suspend } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \langle \bar{h}', \text{weak suspend } \mathcal{S}' \text{ when } (\sigma), \mathcal{A}, \mathcal{C} \rangle} \\
(s_3) \quad \frac{(\forall c \in \mathcal{C}. c \succeq C_S) \wedge C \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak suspend } \mathcal{S} \text{ when } (\sigma) \rangle \xrightarrow{c} \left\langle \bar{h}', \begin{array}{l} \text{weak immediate suspend} \\ \mathcal{S}' \\ \text{when } (\sigma) \end{array}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(s_4) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true} \wedge (\forall c \in \mathcal{C}. c \succeq C_S) \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\left\langle \mathcal{E}, \bar{h}, C_S, \begin{array}{l} \text{weak immediate} \\ \text{suspend} \\ \mathcal{S} \\ \text{when } (\sigma) \text{ reset } \mathcal{S}_R \end{array} \right\rangle \xrightarrow{c} \left\langle \bar{h}', \begin{array}{l} \text{weak immediate} \\ \text{suspend} \\ \mathcal{S}_R \\ \text{when } (\sigma) \text{ reset } \mathcal{S}_R \end{array}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(s_5) \quad \frac{(\exists c \in \mathcal{C}. c \prec C_S \vee \llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false}) \wedge C \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\left\langle \mathcal{E}, \bar{h}, C_S, \begin{array}{l} \text{weak immediate} \\ \text{suspend} \\ \mathcal{S} \\ \text{when } (\sigma) \text{ reset } \mathcal{S}_R \end{array} \right\rangle \xrightarrow{c} \left\langle \bar{h}', \begin{array}{l} \text{weak immediate} \\ \text{suspend} \\ \mathcal{S}' \\ \text{when } (\sigma) \text{ reset } \mathcal{S} \end{array}, \mathcal{A}, \mathcal{C} \right\rangle} \\
(s_6) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \wedge C = \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle \bar{h}', \mathcal{S}', \mathcal{A}, \mathcal{C} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate suspend } \mathcal{S} \text{ when } (\sigma) \text{ reset } \mathcal{S}_R \rangle \xrightarrow{c} \langle \bar{h}', \text{nothing}, \mathcal{A}, \mathcal{C} \rangle}
\end{array}$$

Fig. 4.8. Transition Rules VI (Weak Suspension)

```

weak suspend
  weak immediate abort
  S
  when (true) ;
  pause (C_S) ;
  S
  when (σ) ;

```

Hence, the first step of \mathcal{S} related to clock C_S is executed. Then, the abortion takes place and the control flow will end at the `pause` statement. After one step of the suspension statement is executed, it is also changed back to the `immediate` version. Finally, after one step, the same situation as before will hold. This behavior can also be expressed by SOS rules. However,

there is a drawback of this version: the evaluation of the suspension condition σ needs to be done in the first instant of a step of clock C_S . As already explained for the rules for weak abortion, one advantage of the rules is that the condition is only needed at the end of the step to decide where to proceed. This is similar to the rules for pure QUARTZ and should also be possible in this extension. Therefore, this solution would be feasible if only logical correctness would be intended, but for reflecting a causal execution, another solution should be chosen. A second solution would be possible, if the statement \mathcal{S} could be split into $\mathcal{S}_1; \mathcal{S}_2$ where \mathcal{S}_1 is exactly the first step according to the clock C_S . In this case, the rules could simply handle the statement:

```

weak suspend
   $\mathcal{S}_1$ 
  if ( $\sigma$ )
     $\mathcal{S}$ 
  else
     $\mathcal{S}_2$ 
when ( $\sigma$ );

```

Thereby, the first step is executed for sure. Afterwards, depending on σ either the remaining \mathcal{S}_2 is executed in the suspension block, or the whole statement \mathcal{S} is considered again. Note that the condition σ needs to be evaluated at the end of the step of clock C_S . However, since splitting a statement into steps is not trivial, this possibility is not a solution because the definition of semantics should be close to the source code. However, these solutions show that sometimes the statement needs to be executed more than once (when the suspension condition holds), and therefore when the decision should be taken at the end of the step, the initial statement needs to be kept.

For the transition rules, the suspension statement is extended in that it is able to store the last statement when a step of the statement's clock began. Therefore, the following statement and equality is introduced:

weak immediate suspend	weak immediate suspend
\mathcal{S}	$:\equiv$ \mathcal{S}
when (σ) reset \mathcal{S}	when (σ)

The rules are defined for the new introduced statement:

- The weak suspension statement stops the execution in the last instant which belongs to the step of the statement's clock. Rule (s_4) identifies the last instant by the **pause** statements which are reached: If all clocks are higher or equal to the clock, no smaller instants are executed for this step. If also the suspension condition holds in this instant, the execution is stopped and the statement is reset to \mathcal{S}_R which is the initial statement before the current step. Note that also the case is covered that the block is completely consumed but the suspension condition holds.
- Rule (s_5) covers the case where the execution can simply proceed. This is the case if either the suspension condition is evaluated to **false** or an instant where the suspension does not take place is executed. Note that the statement \mathcal{S}_R is exchanged by the current one. This indicates the initial statement of the current step. If a suspension later takes place for this step, the control flow position can be reset by this statement.

- Finally, Rule (s_6) covers the case where the body of the suspension statement is consumed, and the suspension condition does not hold. In this case, the whole suspension block is also consumed.

Finally, rewriting the suspension statement in a way that it stores its own entry point allows one to define the transition rules for this statement. The rewriting is also possible with other solutions, but those require either some information at the beginning of a step, or a deeper analysis of the statements. Thereby, this first solution would not fit with the intended causality, which should be reflected by the semantics. The second solution needs more complex computations, and it cannot be decided locally without taking care of the substatements.

4.4 Reaction Rules

The transition rules are based on a given environment, and they determine the transformation of the source code for the next instant. Furthermore, they determine the clocks of reached `pause` statements. The purpose of the reaction rules is to determine the environment for the execution of an instant. Thereby, the reaction rules start with very low information, which is given by a partial environment containing e. g. values of input variables and values given by delayed assignments of a previous step. Based on this information, the reaction rules determine the assignments being definitely executed, as well as the possibly executed assignments. Based on these assignments, the environment can be updated to get more values known for the variables. The rules are then applied again to gather even more known values. This iteration stops when the environment does not change anymore, hence, when as much as possible values are known. This approach is equivalent to the reaction rules of pure QUARTZ, but the rules differ. In the following, the general form of the rules is explained first, before all rules are given and explained for all statements.

4.4.1 General Form of the Rules

Like for pure QUARTZ, the reaction rules also define a fixpoint computation: they are applied multiple times and each time the environment is updated based on the actions which are collected for execution. Thereby, the reaction rules of the extension are of the following form:

$$\langle \mathcal{E}, h, C_S, \mathcal{S} \rangle \xrightarrow{c} \langle h', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle$$

The rule specifies that during execution of an instant of clock c of the statement \mathcal{S} given the variable environment \mathcal{E} , the actions $\mathcal{A}^{\text{must}}$ are executed and the `pause` statements of the clocks $\mathcal{C}^{\text{must}}$ are reached. The actions \mathcal{A}^{can} are possibly executed and `pause` statements with the clocks \mathcal{C}^{can} are possibly reached. The meaning of the individual symbols are described in the following.

- *Instant Clock c*
As for the transition rules, the clock of the currently processed instant is c .

- *Environment \mathcal{E}*
 \mathcal{E} represents the environment containing the values known so far for this instant. Thereby, \mathcal{E} may only be partially defined, because the values of some variables are not yet determined, or the values will be defined in a later instant.
- *Incarnation Levels \bar{h} and h'*
 Both symbols for the incarnation level are used here in the same way as for the transition rules. Thereby, \bar{h} is the incarnation level *before* the statement, i. e. it counts all scopes entered while reaching this statement, and h' is the updated incarnation level *after* the statement, i. e. it is updated by the scopes additionally entered during processing \mathcal{S} .
- *Statement Clock $C_{\mathcal{S}}$*
 Like for the transition rules, the least clock being visible at the current position in the program is given by $C_{\mathcal{S}}$. The clock is updated, whenever a clock declaration is entered.
- *Possibly Executed Actions \mathcal{A}^{can}*
 The set \mathcal{A}^{can} contains the actions which are possibly executed in the considered instant based on the (partial) environment.
- *Definitely Executed Actions $\mathcal{A}^{\text{must}}$*
 The set $\mathcal{A}^{\text{must}}$ contains the actions which are definitely executed in the considered instant based on the (partial) environment.
- *Possibly Reached Clocks Actions \mathcal{C}^{can}*
 The set \mathcal{C}^{can} contains the clocks of the **pause** statements which are possibly reached by an execution of the considered instant based on the so far known environment.
- *Definitely Reached Clocks Actions $\mathcal{C}^{\text{must}}$*
 The set $\mathcal{C}^{\text{must}}$ contains the clocks of the **pause** statements which are definitely reached by an execution of the considered instant based on the so far known environment.

The reaction rules for pure QUARTZ encode the information about the possibly and definitely reached **pause** statement in two flags, because there, it is only of interest whether a **pause** statement is reached or not. For the extension, the clocks of the **pause** statements are of interest and therefore, they are collected here. The actions are collected in the same way as for pure QUARTZ, the sets have the relation $\mathcal{A}^{\text{must}} \subseteq \mathcal{A}^{\text{can}}$, since each action that must be executed can also be executed. The same holds for the clocks: $\mathcal{C}^{\text{must}} \subseteq \mathcal{C}^{\text{can}}$. The clock $C_{\mathcal{S}}$ and also the sets \mathcal{C}^{can} and $\mathcal{C}^{\text{must}}$ are only of interest for the rules and are not used later, in contrast to the actions being used for updating the environment. Furthermore, for most of the reaction rules, it is sufficient to know whether the sets \mathcal{C}^{can} and $\mathcal{C}^{\text{must}}$ are empty or not, which is the same information as encoded by the flags in pure QUARTZ. The clocks contained in these sets are used for the weak preemption statements.

4.4.2 Basic Statements

The definitions of the reaction rules for some basic statements are shown in Figures 4.9 and 4.10, and are explained in the following.

- *Rules for Assignments*
 The immediate assignments are handled by Rule (a_1) collecting the assignment in both action sets. The incarnation function is also collected to use the right variables to evaluate

Assignments

$$(a_1) \quad \langle \mathcal{E}, \bar{h}, C_S, x = \tau \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}, \{(x = \tau, \bar{h})\}, \{(x = \tau, \bar{h})\}, \{\}, \{\} \rangle$$

$$(a_2) \quad \langle \mathcal{E}, \bar{h}, C_S, \text{next}(x) = \tau \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}, \{\}, \{\}, \{\}, \{\} \rangle$$

Time Consuming Statements

$$(p_1) \quad \langle \mathcal{E}, \bar{h}, C_S, \text{await clock}(C) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}, \{\}, \{\}, \{C\}, \{C\} \rangle$$

$$(p_2) \quad \frac{c = C}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{immediate} \\ \text{await clock}(C) \end{array} \right\} \right\rangle \overset{c}{\rightsquigarrow} \langle \bar{h}, \{\}, \{\}, \{\}, \{\} \rangle}$$

$$(p_3) \quad \frac{c \neq C}{\left\langle \mathcal{E}, \bar{h}, C_S, \left\{ \begin{array}{l} \text{immediate} \\ \text{await clock}(C) \end{array} \right\} \right\rangle \overset{c}{\rightsquigarrow} \langle \bar{h}, \{\}, \{\}, \{C\}, \{C\} \rangle}$$

Clock Definitions

$$(c) \quad \frac{\langle \mathcal{E}, \bar{h}, C, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{clock}(C) \{ \mathcal{S} \} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}$$

Conditional Statements

$$(i_1) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{if}(\sigma) \mathcal{S}_1 \text{ else } \mathcal{S}_2 \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle}$$

$$(i_2) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_2 \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}_2^{\text{can}}, \mathcal{A}_2^{\text{must}}, \mathcal{C}_2^{\text{can}}, \mathcal{C}_2^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{if}(\sigma) \mathcal{S}_1 \text{ else } \mathcal{S}_2 \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}_2^{\text{can}}, \mathcal{A}_2^{\text{must}}, \mathcal{C}_2^{\text{can}}, \mathcal{C}_2^{\text{must}} \rangle}$$

$$(i_3) \quad \frac{\left(\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \perp \wedge \bar{h}' = \text{Max}(\bar{h}_1, \bar{h}_2) \right) \wedge \left(\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}_1, \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_2 \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}_2, \mathcal{A}_2^{\text{can}}, \mathcal{A}_2^{\text{must}}, \mathcal{C}_2^{\text{can}}, \mathcal{C}_2^{\text{must}} \rangle \right)}{\langle \mathcal{E}, \bar{h}, C_S, \text{if}(\sigma) \mathcal{S}_1 \text{ else } \mathcal{S}_2 \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}_1^{\text{can}} \cup \mathcal{A}_2^{\text{can}}, \{\}, \mathcal{C}_1^{\text{can}} \cup \mathcal{C}_2^{\text{can}}, \mathcal{C}_1^{\text{must}} \cap \mathcal{C}_2^{\text{must}} \rangle}$$

Fig. 4.9. Reaction Rules I (Basic Statements I)

Sequence

$$\begin{aligned}
 (s_1) \quad & \frac{\mathcal{C}_1^{\text{must}} \neq \{\} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \overset{c}{\mapsto} \langle \bar{h}', \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1; \mathcal{S}_2 \rangle \overset{c}{\mapsto} \langle \bar{h}', \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle} \\
 (s_2) \quad & \frac{\left(\begin{array}{l} \mathcal{C}_1^{\text{must}} = \{\} \wedge \\ \mathcal{C}_1^{\text{can}} \neq \{\} \end{array} \right) \wedge \left(\begin{array}{l} \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \overset{c}{\mapsto} \langle \bar{h}_1, \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle \wedge \\ \langle \mathcal{E}, \bar{h}_1, C_S, \mathcal{S}_2 \rangle \overset{c}{\mapsto} \langle \bar{h}_2, \mathcal{A}_2^{\text{can}}, \mathcal{A}_2^{\text{must}}, \mathcal{C}_2^{\text{can}}, \mathcal{C}_2^{\text{must}} \rangle \end{array} \right)}{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1; \mathcal{S}_2 \rangle \overset{c}{\mapsto} \langle \bar{h}_2, \mathcal{A}_1^{\text{can}} \cup \mathcal{A}_2^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}} \cup \mathcal{C}_2^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle} \\
 (s_3) \quad & \frac{\left(\begin{array}{l} \mathcal{C}_1^{\text{must}} = \{\} \wedge \\ \mathcal{C}_1^{\text{can}} = \{\} \end{array} \right) \wedge \left(\begin{array}{l} \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \overset{c}{\mapsto} \langle \bar{h}_1, \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle \wedge \\ \langle \mathcal{E}, \bar{h}_1, C_S, \mathcal{S}_2 \rangle \overset{c}{\mapsto} \langle \bar{h}_2, \mathcal{A}_2^{\text{can}}, \mathcal{A}_2^{\text{must}}, \mathcal{C}_2^{\text{can}}, \mathcal{C}_2^{\text{must}} \rangle \end{array} \right)}{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1; \mathcal{S}_2 \rangle \overset{c}{\mapsto} \langle \bar{h}_2, \mathcal{A}_1^{\text{can}} \cup \mathcal{A}_2^{\text{can}}, \mathcal{A}_1^{\text{must}} \cup \mathcal{A}_2^{\text{must}}, \mathcal{C}_2^{\text{can}}, \mathcal{C}_2^{\text{must}} \rangle}
 \end{aligned}$$

Loop

$$(l) \quad \frac{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\mapsto} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{do } \mathcal{S} \text{ while } (\sigma); \rangle \overset{c}{\mapsto} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}$$

Parallel Threads

$$(p) \quad \frac{\left(\begin{array}{l} \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \rangle \overset{c}{\mapsto} \langle \bar{h}_1, \mathcal{A}_1^{\text{can}}, \mathcal{A}_1^{\text{must}}, \mathcal{C}_1^{\text{can}}, \mathcal{C}_1^{\text{must}} \rangle \wedge \\ \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_2 \rangle \overset{c}{\mapsto} \langle \bar{h}_2, \mathcal{A}_2^{\text{can}}, \mathcal{A}_2^{\text{must}}, \mathcal{C}_2^{\text{can}}, \mathcal{C}_2^{\text{must}} \rangle \end{array} \right) \wedge \left(\begin{array}{l} \bar{h}' = \text{Max}(\bar{h}_1, \bar{h}_2) \wedge \\ \mathcal{A}^{\text{can}} = \mathcal{A}_1^{\text{can}} \cup \mathcal{A}_2^{\text{can}} \end{array} \right)}{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S}_1 \parallel \mathcal{S}_2 \rangle \overset{c}{\mapsto} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}_1^{\text{must}} \cup \mathcal{A}_2^{\text{must}}, \mathcal{C}_1^{\text{can}} \cup \mathcal{C}_2^{\text{can}}, \mathcal{C}_1^{\text{must}} \cup \mathcal{C}_2^{\text{must}} \rangle}$$

Local Declaration

$$(d) \quad \frac{\langle \mathcal{E}, [\bar{h}]_x^{\bar{h}(x)+1}, C_S, \mathcal{S} \rangle \overset{c}{\mapsto} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \{\alpha x; \mathcal{S}\} \rangle \overset{c}{\mapsto} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}$$

Nothing

$$(n) \quad \langle \mathcal{E}, \bar{h}, C_S, \text{nothing} \rangle \overset{c}{\mapsto} \langle \bar{h}, \{\}, \{\}, \{\}, \{\} \rangle$$

Fig. 4.10. Reaction Rules II (Basic Statements II)

the expression later, and to assign the right incarnation of variable x . Since no `pause` statement is reached, the clock sets remain empty. Delayed assignments do not affect the current instant and therefore, they are simply ignored by Rule (a_2) .

- *Rules for Time Consuming Statements*

Instead of the assignments, the time consuming statements affect only the clock sets. An instant ends during the execution when either a `pause` statement is reached, or when an `immediate await clock` statement is reached with a different clock as the one the statement expects. Both of these cases are handled by Rule (p_1) and Rule (p_3) , whereas Rule (p_2) handles the case where the *barrier* is crossed with the right clock.

- *Rules for Local Clock Definitions*

The local clock scopes are handled by Rule (c) applying the reaction rules to the sub-statement with an updated statement clock.

- *Rules for Conditional Statements*

The rules for the conditional `if` statement illustrate very well the meaning of the action and clock sets. If the condition is evaluated to `true` or `false`, the rules (i_1) or (i_2) just select the whole derivation by the result of applying the reaction rules to one of the branches. However, if the condition is evaluated to \perp meaning that not enough information is contained in the environment for selecting one branch, the common information of both branches are combined. Thereby, Rule (i_3) takes the union of the sets $\mathcal{A}_1^{\text{can}}$ and $\mathcal{A}_2^{\text{can}}$ and of the sets \mathcal{C}^{can} and \mathcal{C}^{can} .

- *Rules for Sequence Statements*

In a sequence, the first statement is executed and when it also terminates in this instant, the second one is also executed. Rule (s_1) handles the case that the instant terminates in \mathcal{S}_1 , and Rule (s_3) handles the case that \mathcal{S}_1 is completely executed in this instant. Like for the conditional statement, there is also the case that it cannot be decided yet whether \mathcal{S}_1 is completely executed in this instant or not. Rule (s_2) therefore combines only the information from both statements that is sure.

- *Rules for Loop*

In synchronous languages, instantaneous loops are not allowed, and an instant must end inside the loop body. Hence, the Rule (l) does not need to care about the loop condition and can simply execute the first instant of the loop body.

- *Rules for Parallel Statement*

The reaction rules for the parallel statement are simple. Since both threads are entered in the instant, and the parallel only terminates when both threads terminate, the information obtained by the reaction rules applied to each thread separately can be simply combined.

- *Rule for Local Declaration*

Rule (d) handles the local declarations by updating the incarnation level for the defined variable to execute the substatement. The result of the substatement is provided as result of the whole rule.

- *Rule for Nothing*

Finally, Rule (n) handles the statement `nothing`. Thereby, the rule is defined in a straightforward way.

Besides the new statements for handling **pause** statements of a certain clock, and for handling clock declarations, there is not much difference in the basic rules compared to pure QUARTZ. The constructivity for an instant is very similar to the one for pure QUARTZ and therefore, there is no reason to handle things differently here. Even the check whether the first part of a sequence ends or not is just enhanced to the clock sets, but basically, the same condition is tested. However, the preemption statements, which are considered in the following sections, depend on the actual clocks and there are more differences to the rules of pure QUARTZ.

4.4.3 Strong Preemption

(Strong) Abortion

$$\begin{array}{l}
 (a_1) \quad \frac{\langle \mathcal{E}, \hbar, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \hbar', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, C^{\text{can}}, C^{\text{must}} \rangle}{\langle \mathcal{E}, \hbar, C_S, \text{abort } S \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \hbar', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, C^{\text{can}}, C^{\text{must}} \rangle} \\
 (a_2) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\hbar} = \text{true}}{\langle \mathcal{E}, \hbar, C_S, \text{immediate abort } S \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \hbar, \{\}, \{\}, \{\}, \{\} \rangle} \\
 (a_3) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\hbar} = \text{false} \wedge \langle \mathcal{E}, \hbar, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \hbar', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, C^{\text{can}}, C^{\text{must}} \rangle}{\langle \mathcal{E}, \hbar, C_S, \text{immediate abort } S \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \hbar', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, C^{\text{can}}, C^{\text{must}} \rangle} \\
 (a_4) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\hbar} = \perp \wedge \langle \mathcal{E}, \hbar, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \hbar', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, C^{\text{can}}, C^{\text{must}} \rangle}{\langle \mathcal{E}, \hbar, C_S, \text{immediate abort } S \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \hbar', \mathcal{A}^{\text{can}}, \{\}, C^{\text{can}}, \{\} \rangle}
 \end{array}$$

Fig. 4.11. Reaction Rules III (Strong Abortion)

Again, the difference between the immediate and the delayed preemptions is the treatment of the first step. The rules for abortion and suspension are shown in Figures 4.11 and 4.12. The reaction rules only need to consider the next instant, and the delayed preemption statements only influence the execution after the first step. Therefore, the rules (a_1) and (s_1) for abortion and suspension do not consider the evaluation of the condition and do only consider the execution of the next instant. The immediate version of the preemption are more sophisticated. However, based on the macro definitions introduced in Section 2.2.2, the rules can be directly derived from the definition of the other statements. The rules for the immediate abortion considers the condition in the same way as the **if** statement does. If the condition is evaluated to **true** or **false**, the rules (a_2) or (a_3) select either the behavior of the substatement or simply ignore the substatement. Finally, the Rule (a_4) considers the case the condition cannot be evaluated yet. Like for the **if** condition, the result is combined from both possibilities. The suspension is handled in the same way: the rules (s_2) or (s_3) consider the cases where the further execution is defined by the evaluation of the condition, and Rule (s_4) cannot determine any execution for sure. Remark: the clock C_S added by

Rule (s₄) to \mathcal{C}^{can} is needed because in case of a suspension, the execution just rests until the next step.

(Strong) Suspension

$$\begin{array}{l}
(s_1) \quad \frac{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{suspend } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle} \\
(s_2) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true}}{\langle \mathcal{E}, \bar{h}, C_S, \text{immediate suspend } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}, \{\}, \{\}, \{C_S\}, \{C_S\} \rangle} \\
(s_3) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{immediate suspend } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle} \\
(s_4) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \perp \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{immediate suspend } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \{\}, \mathcal{C}^{\text{can}} \cup \{C_S\}, \{\} \rangle}
\end{array}$$

Fig. 4.12. Reaction Rules IV (Strong Suspension)

4.4.4 Weak Preemption

The difference between strong and weak preemption is that the strong preemption takes place at the beginning of a step, whereas the weak preemption takes place at the end. This means for pure QUARTZ that for the weak version, the data flow is executed before the preemption takes place. However, in the extension, the weak preemption takes place after the last instant of a step of clock C_S , since steps of the clocks defined inside the preemption block are not considered. The reactions rules only consider the next instant and therefore have to decide if a step of clock C_S ends or not. This can be expressed by the following conditions:

1. The step of clock C_S ends for sure in the substatement, hence, a **pause** statement of clock C_S , or lower is reached in this instant:

$$\exists c \in \mathcal{C}^{\text{must}}. c \prec C_S$$

2. The step of clock C_S ends for sure, hence, no **pause** statement of clock C_S , or lower can be reached anymore in this instant:

$$\forall c \in \mathcal{C}^{\text{can}}. c \succeq C_S$$

3. The step of clock C_S possibly ends, hence, it cannot be decided whether a **pause** statement of clock C_S , or lower can be reached or not in this instant:

$$(\exists c \in \mathcal{C}^{\text{can}}. c \prec C_S) \wedge (\forall c \in \mathcal{C}^{\text{must}}. c \succeq C_S)$$

Note that the case that the whole block is finished, hence $\mathcal{C}^{\text{can}} = \{\}$ holds, is covered by the first case, since this case is handled for the weak suspension like a finished step. This point is discussed with the rules below. Together with the three-valued evaluation of the preemption condition, the following cases are possible:

	1. not ended	2. ended	3. possibly ended
$\llbracket \sigma \rrbracket_{\mathcal{E}}^h = \perp$	no preemption	possibly preempt	possibly preempt
$\llbracket \sigma \rrbracket_{\mathcal{E}}^h = \text{true}$	no preemption	preempt	possibly preempt
$\llbracket \sigma \rrbracket_{\mathcal{E}}^h = \text{false}$	no preemption	no preemption	no preemption

These are the cases handled by the reaction rules for the weak preemption statements. The rules for the abortion are shown in Figure 4.13. The delayed abortion is handled by Rule (a_1) which behaves like the rule for strong abortion since it does not influence the first step. The Rule (a_2) handles the case when abortion takes place: the data flow is executed for the last instant but the control flow is aborted. In case the condition can be evaluated to `false`, no abortion takes place as it is handled by Rule (a_3) , where the instant is just executed. Finally, rules (a_4) and (a_5) handle the case where it is not sure whether the execution is aborted or not. For sake of simplicity, this case is split into two rules, but the preconditions cover the cases discussed above. Note that the case in which the whole substatement is finished ($\mathcal{C}^{\text{can}} = \{\}$) is handled by Rule (a_2) and (a_4) , but they behave as the whole statement is completely executed.

After having discussed the rules for the weak abortion, the weak suspension follow exactly the same conditions and the rules are shown in Figure 4.14. Rule (s_1) covers again the simple case of the first step where no suspension is considered. The rules (s_2) and (s_3) handle the cases in which the suspension takes place and finally, the rules (s_4) and (s_5) handle the cases in which it is not yet sure if the suspension takes place. The suspension moves the control flow back to a label where the step started from inside the block. Therefore, the clock C_S must be added to the clock sets. Note also that in case that the whole substatement is finished ($\mathcal{C}^{\text{can}} = \{\}$), suspension can take place as it is handled by Rule (a_2) and Rule (a_4) .

4.5 Program Execution

The definition of the semantics is now completed following the approach of pure QUARTZ by describing an interpreter for executing programs based on the SOS rules.

4.5.1 Interpreter

The general scheme of the interpreter to execute a step is as follows: (1) the step is prepared, i. e. inputs are read and the delayed assignments of the last step are executed, (2) the fixpoint based on the reaction rules is determined, i. e. the reaction rules are applied to compute the sets \mathcal{A}^{can} and $\mathcal{A}^{\text{must}}$ to update the environment, and (3) the transition rules determine the residual statement and the delayed assignments for the next step. The execution of the step fails, if the fixpoint iteration cannot determine the values of all variables, or if a write conflict

(Weak) Abortion

$$\begin{array}{l}
(a_1) \quad \frac{\langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak abort } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle} \\
(a_2) \quad \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true} \wedge (\forall c \in \mathcal{C}^{\text{can}}. c \succeq C_S) \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \{\}, \{\} \rangle} \\
(a_3) \quad \frac{\left(\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \vee \exists c \in \mathcal{C}^{\text{must}}. c \prec C_S \right) \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle} \\
(a_4) \quad \frac{\left(\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \perp \wedge \neg (\exists c \in \mathcal{C}^{\text{must}}. c \prec C_S) \right) \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \{\}, \mathcal{C}^{\text{can}}, \{\} \rangle} \\
(a_5) \quad \frac{\left(\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true} \wedge \exists c \in \mathcal{C}^{\text{can}}. c \prec C_S \wedge \forall c \in \mathcal{C}^{\text{must}}. c \succeq C_S \right) \wedge \langle \mathcal{E}, \bar{h}, C_S, \mathcal{S} \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate abort } \mathcal{S} \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \{\}, \mathcal{C}^{\text{can}}, \{\} \rangle}
\end{array}$$

Fig. 4.13. Reaction Rules V (Weak Abortion)

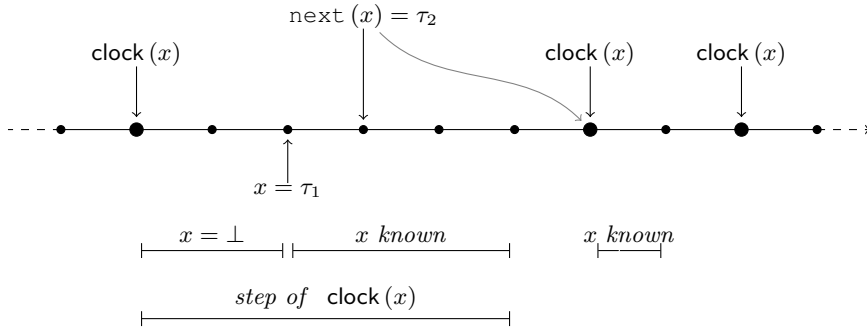
occurs, i. e. a variable is assigned twice with different values. The simulator for the extension works in the same way for each instant, but it has to take care of variables of different clocks and of the clocks itself.

Before the simulator for the extension is explained, consider Figure 4.15 showing a simulation trace with respect to a variable x . Thereby, the big dots are instants of the clock of x , and the small dots represent instances of a lower clock which is not further specified. According to the definition of steps, a step of clock (x) starts with an instant of this clock and ends in the last instant before clock (x) holds again. For a step, the variable x can get a value in three different ways: (1) x can e. g. get a value assigned by an immediate assignment in one of the instants, hence, in a substep. In the instances before the immediate assignment is executed, the value of x is \perp (unknown), and it changes to the actual value with the assignment, hence, x is known for the remaining step. (2) x can also get a value by a delayed assignment specifying the value for the next step (and not for the next instant), hence, this assigned value must be kept until the next step of clock of x starts, i. e. in the next instant of clock of x , and the variable is then known for the whole step. Finally, (3) x can get its default value if it is not explicitly set by an assignment. In this case the value of x from the last step is used if x is a memorized variable, or the default value is used if x is an event. The description about *when* this default reaction sets a value to x is postponed here, and is discussed later. Accordingly to this description, a variable can get its value during the

(Weak) Suspension

$$\begin{aligned}
(a_1) \quad & \frac{\langle \mathcal{E}, \bar{h}, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak suspend } S \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle} \\
(a_2) \quad & \frac{\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true} \wedge (\forall c \in \mathcal{C}^{\text{can}}. c \succeq C_S) \wedge \langle \mathcal{E}, \bar{h}, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate suspend } S \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \{C_S\}, \{C_S\} \rangle} \\
(a_3) \quad & \frac{\left(\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{false} \vee \exists c \in \mathcal{C}^{\text{must}}. c \prec C_S \right) \wedge \langle \mathcal{E}, \bar{h}, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\langle \mathcal{E}, \bar{h}, C_S, \text{weak immediate suspend } S \text{ when } (\sigma) \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle} \\
(a_4) \quad & \frac{\left(\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \perp \wedge \neg (\exists c \in \mathcal{C}^{\text{must}}. c \prec C_S) \right) \wedge \langle \mathcal{E}, \bar{h}, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\left\langle \mathcal{E}, \bar{h}, C_S, \begin{cases} \text{weak immediate suspend} \\ S' \\ \text{when } (\sigma) \end{cases} \right\rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \{\}, \mathcal{C}^{\text{can}} \cup \{C_S\}, \{\} \rangle} \\
(a_5) \quad & \frac{\left(\llbracket \sigma \rrbracket_{\mathcal{E}}^{\bar{h}} = \text{true} \wedge \exists c \in \mathcal{C}^{\text{can}}. c \prec C_S \wedge \forall c \in \mathcal{C}^{\text{must}}. c \succeq C_S \right) \wedge \langle \mathcal{E}, \bar{h}, C_S, S \rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, \mathcal{C}^{\text{can}}, \mathcal{C}^{\text{must}} \rangle}{\left\langle \mathcal{E}, \bar{h}, C_S, \begin{cases} \text{weak immediate suspend} \\ S' \\ \text{when } (\sigma) \end{cases} \right\rangle \overset{c}{\rightsquigarrow} \langle \bar{h}', \mathcal{A}^{\text{can}}, \{\}, \mathcal{C}^{\text{can}} \cup \{C_S\}, \{\} \rangle}
\end{aligned}$$

Fig. 4.14. Reaction Rules VI (Weak Suspension)

Fig. 4.15. Execution Trace for x

step but not necessarily in the first instant, and therefore, it cannot be required that the simulator determines each value in each instant, but the value of a variable must be known until the end of a step is reached. Furthermore, not all values can be reset to \perp when a new instant is simulated, because the value should be kept until the step of the clock of the variable is ended.

The simulator of the extension is also given by a function simulating an instant at each call. Thereby, it gets three environments as inputs and produces three environments as outputs:

- *Values of the last step* \mathcal{E}_{prv}
The environment contains the value of each variable from the last step of its clock, and it is used for the default reaction for memorized variables keeping the old values when no new one is assigned.
- *Values of the current step* \mathcal{E}_{cur}
This environment contains for each variable the value of its current step. It stores \perp when the value is not yet known for the step and get updates by immediate assignments being executed for each variable.
- *Values of the next step* $\mathcal{E}_{\text{next}}$
The values which are assigned to variables by a delayed assignment are stored until the next step of the variable clock starts. Then, the value is transferred to \mathcal{E}_{cur} . The value of x in the simulation trace is e.g. set by a delayed assignment in an instant, but it is updated with the start of the next step, i. e. in the next instant of clock(x).

The interpreter is given in Figure 4.16, and it gets a set of clocks \mathcal{C} , and a statement \mathcal{S} as inputs in addition to the already mentioned environments. Thereby, the set \mathcal{C} contains the clocks of the **pause** statements reached in the previous instant, and \mathcal{S} is the statement the next instant is executed with. For the execution of an instant, the following steps are performed:

1. Based on the clocks contained in the set \mathcal{C} , a clock c for executing the next instant is chosen by the function `ChooseClock` with the property:

$$\forall c' \in \mathcal{C}. c' \succeq c$$

One of the *lowest* clocks contained in \mathcal{C} is used because threads synchronize on common **pause** statements and as long as a lower clock is contained in the set, a **pause** statement of this clock has been reached in the instant before, which means that this substep must be executed before the higher step ends. There can be a choice between unrelated clocks contained in \mathcal{C} . If the chosen clock is the module clock $\mathcal{C}0$, a new step of the module starts and new inputs can be given. In this case, the function `ReadInputs` determines a new environment with the input values for the next step.

2. Based on the chosen clock, a new step starts with the executed instant of clock c . For a new step of clock c , all variables with a clock lower or equal to c have to be reset to unknown (except that there is a delayed assignment from the previous step). The values in \mathcal{E}_{cur} of those variables are therefore transferred to \mathcal{E}_{prv} , since they are now the values of the previous step. Also, the values in \mathcal{E}_{cur} are updated for those variables: if a value can be obtained from a delayed assignment, it is taken, and otherwise the variable is

```

function Instant ( $\mathcal{E}_{\text{prv}}, \mathcal{E}_{\text{cur}}, \mathcal{E}_{\text{nxt}}, C, \mathcal{S}$ )
begin
   $c := \text{ChooseClock}(C)$ 
  if  $c = C_0$  then  $\mathcal{E}_{\text{in}} := \text{ReadInputs}()$  else  $\mathcal{E}_{\text{in}} := \mathcal{E}^\perp$ 
  end if
   $\mathcal{E}_{\text{prv}} := (\mathcal{E}_{\text{prv}})_{/c \not\leq} \dot{\cup} (\mathcal{E}_{\text{cur}})_{/c \leq}$ 
   $\mathcal{E}_{\text{cur}} := (\mathcal{E}_{\text{cur}})_{/c \not\leq} \dot{\cup} (\mathcal{E}_{\text{nxt}})_{/c \leq} \dot{\cup} \mathcal{E}_{\text{in}}$ 
   $\mathcal{E}_{\text{nxt}} := (\mathcal{E}_{\text{nxt}})_{/c \not\leq}$ 
   $\mathcal{h}_{\text{init}} := \{(x, 0) \mid x \in \mathcal{V}\}$ 
  do # fixpoint iteration
     $\mathcal{E}_{\text{old}} := \mathcal{E}_{\text{cur}}$ 
     $\langle \mathcal{h}_{\text{new}}, \mathcal{A}^{\text{can}}, \mathcal{A}^{\text{must}}, C^{\text{can}}, C^{\text{must}} \rangle : \stackrel{c}{\leftarrow} \langle \mathcal{E}, \mathcal{h}_{\text{init}}, \mathcal{S} \rangle$ 
    foreach  $x \in \mathcal{V}^{\text{loc}} \cup \mathcal{V}^{\text{out}}$  do
       $\mathcal{H} := \{\mathcal{h}(x) \mid (x = \tau, \mathcal{h}) \in \mathcal{A}^{\text{can}}\}$ 
      foreach  $i$  in  $0 \dots \mathcal{h}(x)$  do
        if  $i \notin \mathcal{H} \wedge \mathcal{E}_{\text{cur}}^{\mathcal{h}}(x) = \perp$  then
          if  $i \neq \mathcal{h}(x)$  or  $\text{DoDefault}(x)$  do
             $\mathcal{E}_{\text{cur}} := \text{defaultUpdate}(\mathcal{E}_{\text{prv}}, \mathcal{E}_{\text{cur}}, i, x)$ 
          end if
        end if
      end foreach
    end foreach
    foreach  $(x = \tau, \mathcal{h}) \in \mathcal{A}^{\text{must}}$  do
       $\mathcal{E}_{\text{cur}} := [\mathcal{E}_{\text{cur}}]_{(x, \mathcal{h})}^{\llbracket \tau \rrbracket_{\mathcal{E}_{\text{cur}}^{\mathcal{h}}}}$ 
    end foreach
  while  $(\mathcal{E}_{\text{old}} \neq \mathcal{E}_{\text{cur}})$ 
   $\langle \mathcal{S}', \mathcal{h}_{\text{new}}, \mathcal{A}, C \rangle : \stackrel{c}{\leftarrow} \langle \mathcal{E}, \mathcal{h}_{\text{init}}, C, \mathcal{S} \rangle$ 
   $C := C \cup C_0$ 
  if  $\exists x \in \mathcal{V}. (\nexists c \in C. c \prec \text{clock}(x)) \wedge \exists 0 \leq i \leq \mathcal{h}_{\text{new}}(x). \mathcal{E}_{\text{cur}}^i(x) \in \{\perp, \top\}$ 
    then  $\text{Fail}()$ 
  end if
  foreach  $(\text{next}(x) = \tau, \mathcal{h}) \in \mathcal{A}$  do  $\mathcal{E}_{\text{nxt}} := [\mathcal{E}_{\text{nxt}}]_{(x, \mathcal{h})}^{\llbracket \tau \rrbracket_{\mathcal{E}_{\text{cur}}^{\mathcal{h}}}}$  end foreach
   $\mathcal{E}_{\text{cur}} := \{(x, [\mathcal{E}_{\text{cur}}^{\mathcal{h}_{\text{new}}(x)}(x)]) \mid x \in \mathcal{V}\}$ 
   $\mathcal{E}_{\text{nxt}} := \{(x, [\mathcal{E}_{\text{nxt}}^{\mathcal{h}_{\text{new}}(x)}(x)]) \mid x \in \mathcal{V}\}$ 
  # write outputs
  if  $C = \{C_0\}$  then  $\text{WriteOutputs}(\mathcal{E}_{\text{cur}})$ 
  return  $(\mathcal{E}_{\text{prv}}, \mathcal{E}_{\text{cur}}, \mathcal{E}_{\text{nxt}}, C, \mathcal{S}')$ 
end

```

Fig. 4.16. Interpreter

set to \perp . Furthermore, the new input environment is also added defining new values if $c = \text{C0}$, and the values of the delayed assignments which have been transferred to the current environment, are removed from $\mathcal{E}_{\text{next}}$. Finally, the function \tilde{h}_{init} is defined and set to 0 for each variable.

3. Having the environments and incarnation levels prepared, the fixpoint iteration is now started to complete the environment \mathcal{E}_{cur} as much as possible. The iteration terminates if no changes occur anymore for the environment. Therefore, the environment is copied to \mathcal{E}_{old} so that the loop can decide at the end of the iteration if any new change occurred. The reaction rules are then applied to the given statement with \mathcal{E}_{cur} to determine the possibly executed actions. The statement's clock for the rules is C0 since this is the outermost (implicitly) defined clock. The incarnation level is also \tilde{h}_{init} , since the execution starts from outside of any local variable declaration; it is only updated during the application of the rules whenever a new declaration is entered. It results in \tilde{h}_{new} containing for each variable the number of the new declaration blocks entered during this instant. Based on the obtained information, the environment is then updated.
4. The variables which are not explicitly set in a step, should get their default values by the default reaction. In pure QUARTZ, this is done when no action for writing a variable is contained in the set \mathcal{A}^{can} . However, this cannot be only decided for the extension by the set \mathcal{A}^{can} , since instants and steps do not coincide. The variable could be written in a later instant of the same step. Hence, it must be determined that a variable (1) is not written in this instant, and (2) it is not written until the end of the step. The first concern can be decided by the set \mathcal{A}^{can} , whereas the second one can be only decided by *looking ahead* of the currently simulated instant. The reaction rules also determine the resulting incarnation level \tilde{h}_{new} counting for each variable how often its scope is entered in the instant. If $\tilde{h}_{\text{new}}(x) = 0$ holds for a variable x , then no scope was entered for the variable. This is e.g. the case for all input and output variables. For local variables, whose scope was entered one or more times, $\tilde{h}_{\text{new}}(x) > 0$ holds. Practically, there are some values $\mathcal{E}_{\text{cur}}^0(x), \dots, \mathcal{E}_{\text{cur}}^{\tilde{h}_{\text{new}}(x)}(x)$ of the variable x used in this instant, where $\mathcal{E}_{\text{cur}}^0(x)$ is the value of the variable whose scope was just left, $\mathcal{E}_{\text{cur}}^i(x)$ with $i = 1, \dots, \tilde{h}_{\text{new}}(x) - 1$ are the values of the variables whose scope was entered *and* left, and finally $\mathcal{E}_{\text{cur}}^{\tilde{h}_{\text{new}}(x)}(x)$ is the value of the variable whose scope was just entered.

The implementation works as follows. For each variable x , the set \mathcal{H} is determined containing the incarnation levels of x , whose assignments are contained in \mathcal{A}^{can} . The next loop iterates through all incarnation levels of x and considers those which are not contained in \mathcal{H} , but with the value \perp assigned by \mathcal{E}_{cur} . Note that these are exactly the preconditions of the default action of pure QUARTZ so far. Now it can be distinguished between the variables whose scope was left and the others. If the scope was left, it is sure that no action will set them in a later instant. Hence, the default reaction can assign a value for them. These are the variables with the levels $0, \dots, \tilde{h}_{\text{new}}(x) - 1$. The remaining incarnation level, $\tilde{h}_{\text{new}}(x)$ represents the variable whose scope was either entered, or it is an output variable. The variable is kept and can be possibly set by an assignment in a later instant. This decision, whether it will be set until the end of this step is left to the function `DoDefault` which discussed below. If this function tells the simulator that the variable is not set in this step, the default reaction can be executed for the variable, and it is defined as follows:

$$\text{defaultUpdate}(\mathcal{E}_{\text{prv}}, \mathcal{E}_{\text{cur}}, i, x) = \begin{cases} \text{default}(x) & , \text{if } x \text{ is event} \\ \text{default}(x) & , \text{if } i > 0 \\ \mathcal{E}_{\text{prv}}(x) & , \text{else} \end{cases}$$

- If the default reaction is executed for an event, it is reset to its default value. Otherwise, it is executed for a memorized variable, and there, two cases have to be distinguished: (1) the scope of a new variable was entered, and in this case the variable is reset to its default values, (2) otherwise, the old value from the previous step is taken. Note again, that incarnation level 0 means that this variable was also present in the last step, and an incarnation level > 0 means that the scope of this incarnation is entered in this instant.
5. The actions contained in $\mathcal{A}^{\text{must}}$ are definitely executed, hence, the values they define for a variable are added to the environment. Note that the actions are evaluated with the incarnation function collected during the reaction rules, because it describes the variables which are visible at this point in the source code.
 6. After the values for the current instant are determined, the next instant can be prepared. Therefore, the transition rules are applied to determine the residual statement, the clocks for the next instant, and also the delayed assignments. The module clock $C0$ is also explicitly added to the clocks for the next instant for two reasons: (1) usually a QUARTZ system does not terminate and it will proceed *stuttering* when its end is reached, and (2) it simplifies the following a little bit when at least one clock is in this set. However, besides the stuttering behavior, this does not change the behavior of the program, since smaller clocks are (according to the semantics) preferred by `ChooseClock`. Furthermore, the transition rules also check that all variables needed to evaluate control-flow conditions, and to evaluate the assignments, are known and have real values. However, at the end of a step of a clock, it must be ensured that all variables of this (or any lower) clock are known. This is ensured by the check, which forces the simulation to fail if this is not the case.
 7. The following instant can now be prepared by updating the environment with the delayed assignments collected by the transition rules. And finally, the values of the incarnations are *transferred*, but this needs explanation. Entering a scope means also that the scope of another incarnation of the same variable is left, hence, the next instant will be only in the scope of the new variable. According to the incarnation levels, the value for the new variable is the last one in the list stored by the environments. Therefore, the value list of the environments are reduced to only hold the *last* value. The delayed assignments are handled here in the same way as the immediate assignments, because delayed assignments are also collected during several instants until the next step of the variable starts.
 8. Finally, if a step of the module clock was finished with the simulated instant, the output values are provided to the environment. Then, the interpreter returns the result of the simulated instant. This result can be directly given to the interpreter to simulate the next instant. The module clock was explicitly added to \mathcal{C} , which is consistent with the stuttering behavior when the end of the program is reached. If one wants to check whether the program was finished, he or she can do this by the residual statement \mathcal{S}' , which must be **nothing** for termination.

Finally, an instant of clock c is completed, and the next instant can be executed by calling the interpreter again with the result from the previous call.

4.5.2 Constructive Execution

There are two open discussions left for the interpreter, the function `ChooseClock`, and the function `DoDefault`, both discussed in the following.

Consider first the non-deterministic choice of the clock of the next instant, which is done by the function `ChooseClock` selecting one of the smallest clocks from \mathcal{C} . If there are several choices, taking the wrong one can lead to an error in the simulation, whereas the other one leads to a valid execution. As an example, consider the program shown in Figure 4.17 with the unrelated clocks `C1` and `C2`, and the variable `x` defined on clock `C0`. Assume that the first instant (of clock `C0`) was executed, and the next one (starting from labels `l1` and `l3`) is now considered by the interpreter. Hence, the set of clocks given to the interpreter is $\mathcal{C} = \{C0, C1, C2\}$, and the interpreter can either choose `C1` or `C2`. Obviously, the value of `x` is \perp before this instant, and if the interpreter chooses `C2`, the execution will fail, since the assignment cannot be evaluated. However, if it chooses `C1`, the execution will work. Hence, the execution based on the clocks of instances must be *scheduled* to get the desired behavior. However, even if the wrong choice was made, and the interpreter fails, it could also try another choice, but trying out is not in the spirit of constructive execution. However, by also defining the default reaction, the intention is to keep the scheduling as simple as possible.

Consider now the estimation of the function `DoDefault` to determine whether a variable is set in the further execution of the step of its clock or not. Obviously, to know exactly whether a variable will be set in the following execution of the step or not, requires to look ahead. This can be done by a simple analysis, or by trying out each possibility for execution. However, two interesting over-approximations for this function are defined in the following, both leading to practical results:

- The first approximation of the function can be described with: *a variable x can only be immediately written in an instant of clock (x)* . Hence, the variable can only be written in the first instant of its step, but not later. The question, whether the variable will be written in the remaining step is therefore answered with *no*, and the function can be defined as:

$$\text{DoDefault}(x) = \text{true}$$

Hence, the default reaction will, if necessary, define the value of a variable in the first instant of its step. Note that this does not influence, when and how variables are read, and it also does not forbid to write the variable with a delayed assignment from a lower instant. Obviously, this restriction does not impose any scheduling dependencies, since the semantics forces synchronization by the clocks when values of variables are exchanged. This restriction also leads to a behavior which is very similar to oversampling in `SIGNAL`, and the relation will be discussed in Section 7.2.3.

- The second approximation is similar to the first one, but it forces the restriction only for variables which are used for the communication between unrelated clocks. Hence, if a variable x is read in instances of a clock c , with clock $(x) \prec c$, then it is not (immediately) written in instance of a clock c' with $c \# c'$. Note that delayed assignments are allowed and communication between unrelated clocks is synchronized by the `pause` statements of common clocks.


```

int x;
clock(C1) {
  11: pause(C1);
  x = 5;
  12: pause;
} || clock(C2) {
  13: pause(C2);
  o = x;
  14: pause;
}

```

Fig. 4.17. Example: Parallel Causality with Unrelated Clocks

The first definition can be checked rather simple by a syntactical test on the source code. However, the second version is harder to check [GeBS11a], but, in the same way constructivity can be approximated by e. g. checking for cycles, a check is presented in the next chapter which can be applied after compilation to determine the constructive programs. Also this check is only an abstraction of the exact definition, but since checking the exact causality is very inefficient, it is a practical solution.

Furthermore, the constructivity defined has to look into the future to decide whether a variable will be written in a step or not. However, if the program is proved to be constructive, the synthesis can handle this very efficiently, since in the real target language, the value \perp is not modeled. Instead, it is ensured by the translation that a variable is written to its value before the value is used. Therefore, the synthesis can simply keep the old value (for memorized variables) or the default value (for events) at the beginning of a step. The analysis ensures that the right value is used.

4.6 Summary

This chapter defined the semantics of refined clocks with two sets of SOS rules like the semantics definition of pure QUARTZ in [Schn09]. Besides many similarities in the rules, the main difference is that the original semantics only considers instances of a single clock, whereas the extension has to deal with different clocks. Those clocks control the point of time when conditions are checked, abortions take place, and also when variables can get a new value. The clocks are also responsible for independent execution, which is not present in this way in pure QUARTZ. A simulation procedure was also defined to execute an instant of the extension using the SOS rules. Thereby, this function is again similar to its counterpart of pure QUARTZ, but it also has some differences. First, a clock for an instant must be chosen, and second the execution of the default reaction must be determined. Two solutions have been proposed, whereas both are scheduling independent, i. e. the success of the simulation does not depend on the clock the simulator chooses. Furthermore, the first one is very simple to check, but the second one is more convenient for programming and some abstractions can be also checked in a simple way. This is completely in the mood of checking the constructivity of pure QUARTZ, where it is hard to ensure it for a general program, in most cases a simple

check can also do the job. However, if the program is checked to be constructive, the synthesis can produce nice code for execution.

Chapter 5

Compilation

The original compilation algorithm for QUARTZ translates programs to the intermediate format AIF, which provides a reasonable abstraction from the source language and which serves as a starting point for synthesis and analysis tools. In this way, the computational model and the program behavior is kept but the complex interaction of control-flow statements is removed. This leads to simpler synthesis tools, which only need to focus on their target language. The compile algorithm for QUARTZ was already mentioned in Section 2.2.6 and it is explained in detail in [Schn09]. Even though this is just one way to translate synchronous programs to a target language, the intermediate format has proved to be a useful abstraction and this approach is also kept for the extension considered in this thesis. However, AIF needs to be extended to cover programs with refined clocks.

The intention of this chapter is to present a compilation algorithm together with a new extended intermediate format, which is much simpler than the source language and which can be translated to target languages such as hardware description languages with reasonable effort. To achieve this goal, the presented compilation algorithm will only work on a subset of extended QUARTZ programs, but these restrictions will be clearly motivated. The evaluation of some examples in Chapter 7 will later also show that these restrictions do not have a significant negative impact on practicability. Preliminary versions of the algorithm have been discussed in [GeBS10b, GeBS11].

The chapter is structured as follows. The extension of AIF is first motivated by small programs represented by the intermediate format. Then, the idea of the translation of more complex statements is discussed, before the actual compilation algorithm is defined. Finally, based on the intermediate format, a simple method to check the abstraction of constructivity for dependencies across instants is presented.

5.1 Extended Intermediate Format

The extended intermediate format, which is used as target for the compile algorithm is presented in this section. The explanation focuses on the differences to the original format and does not consider implementation details.

5.1.1 General Idea

The section sketches the representation of simple programs by the intermediate format which is still based on guarded actions representing the data flow and the control flow. Each data-flow assignment in the source code is translated to a guarded action where the guard describes when the action is executed. Hence, the guard expresses the condition under which the assignment is *reached* during program execution. Reaching a certain point during execution is based on the control flow whose states are encoded by the `pause` statements. As already mentioned in Section 2.2, in the original intermediate format, the labels of `pause` statements are used as Boolean signals to identify the control-flow locations in the source code. The labels are active for the steps starting from the related `pause` statements, and they must be explicitly set for the steps where they should hold because they are automatically `false` in the other instants. Technically, they are considered as Boolean events in the intermediate format. The guarded actions obtained from the original compiler have the invariant that at least one label has a positive occurrence in the guards. Those labels identify the control-flow locations where the actual action can be reached from in an instant where the label is active. As an example, consider the following short program on the left-hand side together with the guarded actions on the right-hand side obtained by the original compile algorithm:

<pre> 11: pause; if (γ) x = τ; 12: pause; </pre>	<pre> $\gamma \wedge 11 \Rightarrow$ x = τ 11 \Rightarrow next (12) = true </pre>
--	--

The assignment to the variable `x` can be reached in an instant where the control flow is at label 11 (the previous step ended at the related `pause` statement) and where the condition of the `if` statement is `true`: thus, the guard of the assignment is $\gamma \wedge 11$. In the same instant, the control flow moves from label 11 to label 12, hence 12 will hold in the next instant, this is expressed by the other guarded action representing the control flow. Since 11 is not set again for the next step, it is automatically reset. Consider now a variation of the example where an `await` statement is used instead of the `pause`:

<pre> 11: await (α); if (γ) x = τ; 12: pause; </pre>	<pre> $\gamma \wedge \alpha \wedge 11 \Rightarrow$ x = τ $\neg \alpha \wedge 11 \Rightarrow$ next (11) = true $\alpha \wedge 11 \Rightarrow$ next (12) = true </pre>
--	--

With this change, the further processing of the instant is also based on the condition of the `await` statement. Clearly, the label 11 is only left when the condition α holds. Since labels are considered as events, they have to be set in each instant they should hold and an additional control-flow action ensures that the label 11 is set again as long as α does not hold. The assignment to `x` is then only executed when the `await` statement is left and the first branch of the `if` statement is taken.

Having the semantics as introduced in Chapter 4 for the `await clock (c)` statements in mind, the `pause (c)` statement *waits* for the occurrence of a certain clock signal to execute the next instant (of the clock `c`). Hence, the assignments which are reached from a `pause`

statement are executed when (1) the label holds and (2) the related clock ticks. The `pause` statements *can* be translated to guarded actions similar to the ordinary `await` statements. Consider the following example with refined clocks:

<pre> 11: pause; if (γ) { x = τ₁; 12: pause (C1); y = τ₂; } 13: pause; </pre>	<pre> γ ∧ 11 ∧ C0 ⇒ x = τ₁ 12 ∧ C1 ⇒ y = τ₂ γ ∧ 11 ∧ C0 ⇒ next (12) = true ¬γ ∧ 11 ∧ C0 ⇒ next (13) = true 12 ∧ C1 ⇒ next (13) = true 11 ∧ ¬C0 ⇒ next (11) = true 12 ∧ ¬C1 ⇒ next (12) = true </pre>
---	--

The assignment to `x` is executed when the control-flow is at label `11`, an instant of clock `C0` is executed, and additionally the `if` condition holds. Thereby, the signal `C0` represents the instant clock and `11` holds as long as the control-flow is at the related `pause` statement. In contrast to the translation of the ordinary `await` statement waiting for the occurrence of a certain condition, no control-flow action is explicitly added which *keeps* the label when the related clock does not occur (the last two guarded actions are omitted). The behavior there is encoded in the semantics of the label: a label is only disabled (reset to `false`) after an instant of its clock. In this way, the behavior is partially encoded in the model instead of explicit actions. However, each backend tool has to take care of it. Also, this translation is compatible with the original AIF in the way that a normal AIF can be translated to the refined clocks representation by simply adding the clock `C0`, which is the clock of all labels in pure QUARTZ, to the guard of each guarded action.

5.1.2 Labels and Clocks

In the way labels are introduced in the intermediate format above, they only hold for an instant of the according clock and not for the whole step. The clocks are also only active for one instant. But there is a difference between both. Labels can be set active before the actual instant is executed and they are reset after the instant, whereas clocks only hold during this instant. Hence, labels represent the positions in control flow which have been reached and the clocks are events to proceed the execution. This is similar to the definition of the semantics in Chapter 4 where the `immediate await clock` marks the positions which have been reached before, but the execution only proceeds when the clock is also active.

5.1.3 Local Declarations

Local variable declarations are a fundamental feature in many programming languages and they are also present in QUARTZ. The lifetime of a variable is usually bound to the scope where the variable is visible. The intermediate format AIF abstracts from the control-flow statements and also from the scopes of variables: each variable is globally defined. The lifetime of the variables is the same as the lifetime of the whole system. There is no dynamic variable creation or deletion, which is not desired since also a translation to hardware is

of interest. The compilation procedure must map the original behavior of scopes of local variables to the intermediate format. Hence, the entrance and exit of a variable's scope must be imitated somehow.

A particular characteristic for local declarations in synchronous languages was already mentioned in Section 2.2.3. *Schizophrenia* problems occur when the scope of a local variable is left and re-entered in the same macro step. The consequence for the compilation of refined clocks is discussed later, whereas the introduction here covers only the basic case to imitate the entrance of a variable's scope in the intermediate format. As already explained in Section 2.2.5, the original AIF uses the so-called *default reaction* for this purpose. It is defined as an expression whose value is used for the variable if it does not explicitly get a value by an assignment in the macro step. Thereby, the value of the expression is determined in the previous macro step: it *transfers* a value from the last step to the current one. The default reactions can be used to solve the problems for QUARTZ but they will not work for the extension. The reason for this is that the expression needed to define the default reaction would use variables of lower clocks resulting in possibly several values from the previous step. For transferring the value from the last step to the current one, it cannot be decided which value to take. Also this point is considered later.

The intermediate format introduced here uses a new construct to imitate the entrance of a variable's scope. Therefore, instead of using a default expression to transfer values from the last step, a special *reset* condition

$$\text{reset}(x) = \tau$$

is used for a local variable x to define the instant when it should be *potentially* reset to its default value. This defines the start conditions of the control-flow for the entry of the scope of x . The condition is of Boolean type and has the additional property that it only holds in an instant of the clock of x , i. e.

$$\text{reset}(x) \rightarrow \text{clock}(x).$$

Hence, it can only hold in the first instant of a step of clock of x . This makes the information about the scope entrance available for the whole step and the variable can be reset if it is needed. The advantage of this expression over the original default reaction is that there is no value needed from the previous step, but it is only specified when the default value should be set.

Another property of the reset expression and its impact on the semantics of the intermediate format is that it gains priority over delayed assignments. Hence, when the reset conditions tells to reset a variable then it is done without respect of delayed assignments of the previous step. A value of a variable for a step is consistent to the following evaluation:

$$x := \begin{cases} \tau & , \text{if action } \gamma \Rightarrow x = \tau \text{ is executed in this step} \\ \text{default}(x) & , \text{if } \text{reset}(x) \text{ holds in first instant of step} \\ \tau & , \text{if action } \gamma \Rightarrow \text{next}(x) = \tau \text{ was executed in previous step} \\ \text{default}^+(x) & , \text{else} \end{cases}$$

Note that the definition does not invalidate write conflicts which occur when two different values are set to a variable within one step. Write conflicts have to be checked and omitted.

If none occurs during execution, the above evaluation determines the value. The definition of the value of a variable x seems to be arbitrarily, but it is not. If the reset condition holds in the first instant, it is clear that the scope of a variable is entered in this step. A delayed assignment from a previous step cannot assign this variable, because it has not been visible in the last step. The only assignment the variable can get a value from is an immediate assignment of the current step. In this way, this definition will help to imitate the behavior of the variable's scope for compilation. The second advantage of this definition is that it translates very well to target languages as it will be shown later.

5.1.4 Complete Structure

<pre> module P(int ?i, int !o) { loop { clock(C1) { int x; l1: pause(C1); x = i; l2: pause(C1); o = x; next(x) = o + 1; } l0: pause; } } </pre>	<pre> C0 ∧ st ⇒ next(l1) = true C1 ∧ l1 ⇒ next(l2) = true C1 ∧ l2 ⇒ next(l0) = true C0 ∧ l0 ⇒ next(l1) = true </pre> <p style="text-align: center;"><i>Control-Flow Actions</i></p> <pre> C1 ∧ l1 ⇒ x = i C1 ∧ l2 ⇒ o = x C1 ∧ l2 ⇒ next(x) = o + 1 </pre> <p style="text-align: center;"><i>Data-Flow Actions</i></p> <pre> reset(x) = C0 ∧ l0 </pre> <p style="text-align: center;"><i>Reset Conditions</i></p>
(a) Code Example	(b) Intermediate Format

Fig. 5.1. Example: Extended Intermediate Format

An example of a program represented by the guarded actions in the intermediate format is shown in Figure 5.1. As already mentioned, in the data structure also the variable's declarations, types and the clock tree are stored. The following description focuses on the guarded actions obtained from the compilation procedure. The guarded actions for the labels and assignments are determined as described above, and the reset condition expresses in which instant the scope of the variable x is entered. However, the initial instant ($st \wedge C0$) is not explicitly added because variables have their default value when the module is started. Note that the property that the scope is entered in an instant of clock $C0$ or higher is fulfilled. The delayed assignment to x will never set a value to x because it is executed after the scope of x is left. The reset condition with the above definition ensures that in this case the value of the *new* x is reset.

5.2 Surface and Depth

The original compilation algorithm is based on the distinction between surface and depth of each statement. The surface are all guarded actions which are possibly executed in the step when the statement is started and the depth contains all guarded actions which are executed in later steps, when the control-flow is inside the statement. Since, with the introduction of refined clocks, steps are now related to a certain clock, the definition of surface and depth is also extended: they also refer to a certain clock.

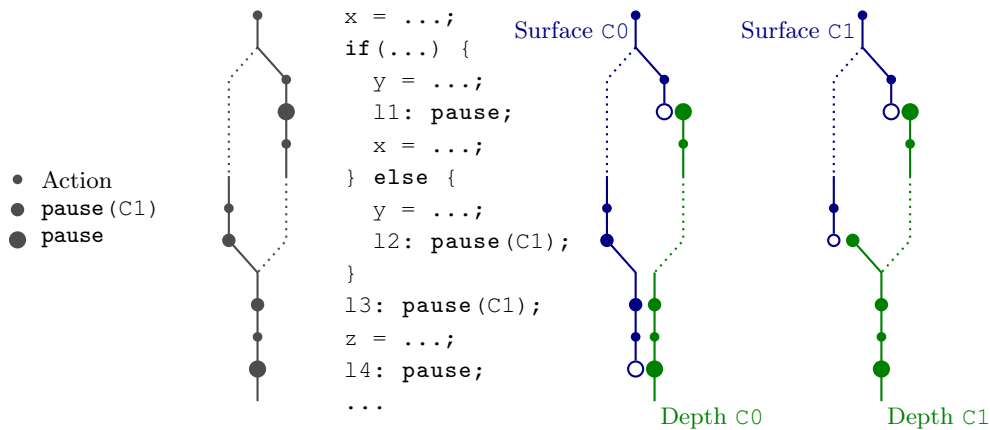


Fig. 5.2. Surface and Depth of Statements

An illustration of surface and depth related to different clocks is given in Figure 5.2. The two clocks C0 and C1 are considered in the example. The surface of clock C0, i. e. the first step of this clock, ends at one of the **pause** statements with label 11 or 14. The depth of clock C0 starts from those labels. On the other hand, the surface of clock C1, i. e. the first step of clock C1, ends at either the label 11 or 12. The depth of clock C1 starts from these labels. Two things can be seen in this example. First, the surface of clock C1 can also end at a label related to clock C0 since at this **pause** statement the step of clock C1 also ends. Second, in the same way an assignment can belong to the surface and the depth in pure QUARTZ, the control-flow, hence the **pause** statements, of a lower clock can belong to the surface and the depth of a higher clock. In this way, the control flow of clock C1 based on the labels 12 and 13 belongs to the surface of clock C0. The label 13 also belongs to the depth of clock C0.

The original behavior of some statements such as loops or preemption is based on the surface and the depth. For refined clocks, this behavior is extended to steps of certain clocks to preserve the abstraction. Therefore, the definitions of surface and depth which are based on certain clocks are important for the translation of QUARTZ with refined clocks to guarded actions.

Instant Surface and Instant Depth

In addition to the surface and depth of a certain clock, the *instant surface* and *depth surface* of a statement are introduced. The instant surface are the guarded actions which are executed in the first instant of the statement, and the instant depth are the guarded actions which are executed in each instant when the control flow already is inside the statement. The instant surface and depth in the example in Figure 5.2 are equivalent to the surface and depth of clock C1. The instant surface ends when the first `pause` statement of any clock is reached. However, the instant surface must not be equal to the surface of a certain clock when new clock declarations or parallel threads are entered. In pure QUARTZ, surface and depth of clock C0 are equal to the instant surface and depth. In this case, there exists no other clock. Finally, if the surface or depth of a certain clock is addressed, the clock is explicitly given, whereas the instant surface and depth is also just referred to with just surface and depth.

5.3 Translation of Certain Statements

This section discusses the translation of several programs using more complicated statements to the intermediate format. It discusses how the programs can be represented by the intermediate format including various options. This will help to understand the actual compile algorithm, which it is presented later.

5.3.1 Control-Flow Graph

Several approaches to represent programs in a more convenient way for automatic analysis than the syntax tree have been published in the past. Control-flow graphs [Alle70] connect so-called *basic blocks* and focus on their control dependence, whereas program dependence graphs also take data dependencies into account [OtOt84, FeOW87]. In contrast to those approaches focusing on sequential threads, Petri nets [Ager79, Petr80a] allow to represent concurrent control-flow and also forking and synchronization of threads. The following considerations are illustrated by another graph showing the conditions for labels to be reached in an instant. Each vertex of the graph simply represents a label in the source code. An edge from a vertex to another one represents a possible transition under a certain condition in an instant from one label to another. The transitions are represented in the intermediate format by control-flow guarded actions. Hence, without completely formalizing the graph representation, it is an illustration of the encoded control flow and will show the effects the compilation will have for several statements.

An example is given in Figure 5.3 showing the source code, the according guarded actions, and the control-flow graph. The edges are only labeled with the conditions which move the control flow from a label to another one. In the expressions, the labels and the clocks are omitted since they are determined by the vertex, e. g. the vertex from l0 to l1 is only labeled by γ_1 instead of $C0 \wedge l0 \wedge \gamma_1$, since the first part of the guard is already determined. In general, because of the parallel statement it is possible that more than one label can be active at the same point of time. This behavior is also reflected by the control-flow graph. Hence, each label is represented by an own node in contrast to the EFSM, where all combinations of reachable labels are combined to a state. The control-flow graphs are used in the following

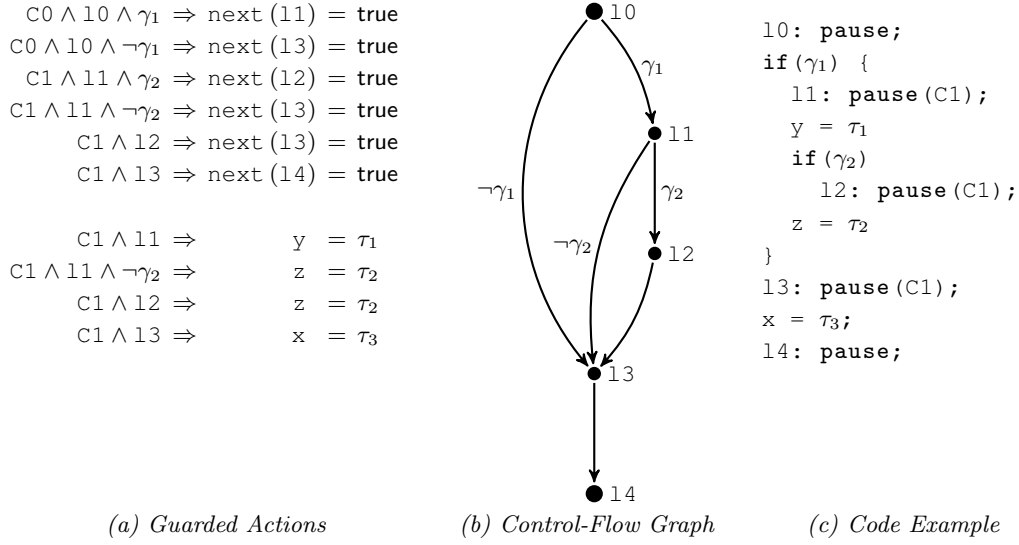


Fig. 5.3. Examples: Control-Flow Graph

as an illustration for the control flow encoded by guarded actions. With their help, the effect of statements during compilation of the control flow can be shown more intuitively.

5.3.2 Parallel Threads

The parallel statement defines simultaneously executed threads, which are started together in the same instant and which will be synchronized at the end. Hence, other statements after the parallel statement are only executed when both threads terminated. Additionally, the threads synchronize on `pause` statements of common clocks (but they do not synchronize on `pause` statements of clocks which are locally declared).

Consider the example given in Figure 5.4 having one refined clock, which is locally defined in the first thread of the parallel statement. Both threads are started in the same instant from label 10, and the label 16 is reached in the instant where the last thread terminates (or where both terminate together). The control flow of both threads during their execution is independent of each other, but they will synchronize on `pause` statements of common clocks (which is only `C0` in the example). The synchronization of common clocks is done by the semantics of the intermediate format: steps of the same clock are executed together. Therefore, the threads can be generally translated independently to the intermediate format. However, there is still the question how *fork* and *join* are represented by guarded actions. The fork is simple since both threads are started together in the instant the parallel statement is started: they just get the same start condition ($10 \wedge C0$ in the example). More of interest is the join, because it has to take care of different situations options. Therefore, consider the termination condition of the parallel statement $S_1 \parallel S_2$ in pure QUARTZ:

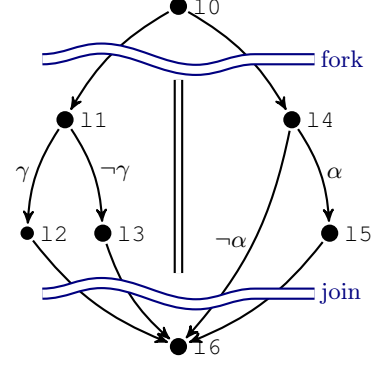
$$\underbrace{\text{term}_{S_1} \wedge \text{term}_{S_2}}_{(a)} \vee \underbrace{\text{term}_{S_1} \wedge \neg \text{insd}_{S_2}}_{(b)} \vee \underbrace{\text{term}_{S_2} \wedge \neg \text{insd}_{S_1}}_{(c)}$$

```

10: pause;
{ // Thread 1
  clock(C1) {
    11: pause;
    if (γ)
      12: pause(C1);
    else
      13: pause;
  }
}
||
{ // Thread 2
  14: pause;
  if (α)
    15: pause;
}
16: pause;

```

(a) Code Example



(b) Control-Flow Graph

Fig. 5.4. Examples: Parallel Execution with Refined Clocks

This condition holds when both threads terminate in the same step (a), or when one thread terminates, and the other one has been already terminated (b, c). The condition $\text{insd}_{\mathcal{S}}$ of a statement \mathcal{S} is a disjunction of all labels which occur in \mathcal{S} . The same condition can also be used for clock refinement and the guarded actions look as follows:

$$\begin{aligned}
10 \wedge C0 &\Rightarrow \text{next}(11) = \text{true} \\
10 \wedge C0 &\Rightarrow \text{next}(14) = \text{true} \\
\gamma \wedge 11 \wedge C0 &\Rightarrow \text{next}(12) = \text{true} \\
\neg\gamma \wedge 11 \wedge C0 &\Rightarrow \text{next}(13) = \text{true} \\
\alpha \wedge 11 \wedge C0 &\Rightarrow \text{next}(15) = \text{true} \\
\underbrace{(12 \wedge C1 \vee 13 \wedge C0) \wedge (\neg\alpha \wedge 14 \wedge C0 \vee 15 \wedge C0)}_{(a)} &\Rightarrow \text{next}(16) = \text{true} \\
\underbrace{(12 \wedge C1 \vee 13 \wedge C0) \wedge \neg(14 \vee 15)}_{(b)} &\Rightarrow \text{next}(16) = \text{true} \\
\underbrace{\neg(11 \vee 12 \vee 13) \wedge (\neg\alpha \wedge 14 \wedge C0 \vee 15 \wedge C0)}_{(c)} &\Rightarrow \text{next}(16) = \text{true}
\end{aligned}$$

Thereby, the action with guard (a) describes the case when both threads terminate together. The guarded action with guard (b) describes the termination when the first thread terminates and the second one has already terminated before. Finally, the third guarded action with guard (c) describes the termination when the second thread terminates and the first one has already terminated before.

5.3.3 Loops and Local Declarations

As already explained, there are no means of dynamic variable creation or deletion in the intermediate format, and variables have the same lifetime as the whole system. For this reason, entering the scope of a variable, which means that the variable is possibly reset according to the semantics, must be handled by the intermediate format, or by the compilation to it, respectively. Furthermore, the issue of schizophrenia has been already introduced in Section 2.2.3, and it is now analyzed in more detail to explain the solution taken for the extension. The main problem here is that the scope of a variable is entered and left in the same step, which can only occur with loops, but due to weak abortion statements, the number of entered scopes depends on the number of nested loops. A solution that was introduced for ESTEREL [Berr97a] transforms a loop statement (the general case in QUARTZ syntax is used here)

$$\text{while}(\sigma) \{ \mathcal{S} \}$$

before compilation by duplicating the body, i. e. the scope of the local variable, to

$$\text{while}(\sigma) \{ \mathcal{S}; \text{if}(\sigma) \{ \mathcal{S} \} \}.$$

Due to renaming of different local variables which are represented by the same variable name, technically, two local variables are now present. The replacement ensures that the scope of one local variable cannot be left and entered in the same step. Instead, the scope of one variable is left, and the scope of the *other* one is entered. In this way, the compiler needs only to consider the entrance of a scope but not the overlapping in the same step. The disadvantage of this approach is that it leads to an exponential blow-up of the code for nested loops. Even worse, the duplication is not enough for QUARTZ, because a delayed assignment can set a value of the local variable when its scope is left. This assignment can affect the value of the local variable because the scope may be entered again one step after. This problem is not present in ESTEREL because delayed assignments are not available there. For QUARTZ, it is not only needed to ensure that the scope of the same variable is not re-entered in the same step, but also not in the following one. Therefore, a third copy of the loop body can be used, which obviously leads to an even bigger blow-up:

$$\text{while}(\sigma) \{ \mathcal{S}; \text{if}(\sigma) \{ \mathcal{S} \} \text{if}(\sigma) \{ \mathcal{S} \} \}$$

The QUARTZ compiler solves this smartly by two approaches: First, delayed assignments which are executed when the scope of a variable is left are *disabled* by an additional condition for their guards. This basically ensures that the third copy is not needed. Second, it duplicates only the first step and not the whole loop body. Hence, only the code of the scope is duplicated which can overlap during execution. Anyway, a copy of the local variable needs to be introduced, which is called an *incarnation* of the variable. The copy is used to represent the behavior of the new scope of x : the surface of the scope is duplicated and a renaming of x in the surface ensures that the incarnation is used instead of the original variable. Delayed assignments to x in the surface are not affected by the renaming, since the original x is used in the following step. Due to nested loops and weak abortion, more than one incarnation of a local variable is also possible in pure QUARTZ.

Summarizing the above, during execution based on the source code, the scope of a local variable can be left and entered. Technically, each time the scope is entered, a new

```

loop {
  bool x;
  ...
10: pause;
  clock(C1) {
    next(x) = ...;
    ...
    while( $\gamma$ ) {
      ...
      11: pause(C1);
    }
    if( $\alpha$ )
      12: pause;
  }
}

```

Fig. 5.5. Example: Reachability of Loop Exit

variable is created, and when the scope is left, the variable is not needed anymore. Since dynamic creation and deletion is not possible in AIF (and also not desired for hardware or for predictable code generation), an upper bound of coexisting variables is determined and translated to AIF. The whole behavior of the original code is then represented based on these variables.

An interesting case, which was already mentioned, is given by delayed assignments to local variables which are executed in the step in which the scope of the variable is left. Hence, the value is assigned to the variable at a point of time where the execution is no longer inside their scope, and technically, the variable does not exist anymore. This can be considered in two different ways from the semantical point of view:

1. Since the variable does not exist anymore in the step where it should get the value, the assignments can be considered as invalid.
2. Since the variable does technically no longer exist, it should be safe to assign any value to it, because it cannot be read anymore. The delayed assignment can be executed, but it will not have an effect.

The QUARTZ semantics and also the semantics of the extension choose the second interpretation, hence it is safe to assign a variable with a delayed assignment whose scope is left. However, the solution of the QUARTZ compiler for delayed assignments is to *disable* them when the scope of the variable is left. In pure QUARTZ, the reachability of each position in source code can be expressed by a Boolean condition holding the whole step. Therefore, the condition for leaving a loop body (resp. a variable scope) can be identified and each delayed assignment in the loop body can be additionally guarded by this condition so that it is not executed in case of leaving the scope. This condition cannot be expressed for the extension as it is shown in the example in Figure 5.5. The (outer) loop body can be left by a step of clock C0 from the labels 10 and 12. If the loop body is left from label 10 depends on the condition α of the `if` statement. The condition is computed in the inner loop during some

substeps. Hence, when the `if` condition is reached, the expression α can be evaluated and it can be decided whether the (outer) loop body is left or not. This decision cannot be taken when the delayed assignment to `x` right after the label `l0` is executed. But, to disable delayed assignments it would be needed to be decided at this point. However, as also mentioned above, the `reset(x)` condition introduced for a variable x has priority to delayed assignments. Hence, the solution for the delayed assignments taken for the extension is to evaluate them, but decide later if the value from the delayed assignment is taken or the default value is set.

The second issue, the entrance of the new scope in the same step, is handled by the QUARTZ compiler by duplicating the surface of the loop. This means for the extension that the surface according to the loop's clock must be duplicated, which means that possibly control-flow labels of lower clocks must be also duplicated.

5.3.4 Strong Preemption

The simpler statements basically influence substatements at their start or end point, but not during the execution itself. For instance, the conditional statement only strengthens the start condition of its branches. More complicated statements such as the preemption statements influences the substatements also during their execution by either suspending or aborting them. Thereby, the preemption only takes place at instances of a certain clock: the clock the preemption statement is defined on. An example program is shown in Figure 5.6, which contains the same code for abortion and suspension. The left-hand side shows the program code, whereas the right-hand side shows the structure of the control flow.

Consider the version with the `abort` first. The clock of the `abort` statement is `C1` since it is the surrounding clock declaration meaning that the abortion takes place after `pause` statements of this clock. Hence, abortion can occur at the labels `l2` and `l4` in an instant of clock `C1`. The other labels of clock `C2` defined inside the preemption block are not influenced. The black part of the diagram on the right-hand side of the figure illustrates the control flow as it would be executed without preemption. It is extended by the abortion block in a way that now the execution can *jump* from inside the block to its end when the condition σ holds: the execution of the block is aborted. Please note that the condition σ can only contain variables which are declared on clock `C1` or higher. Therefore, σ must also hold for the whole step and cannot change during substeps of clock `C2`.

Second, consider the version with the `suspend` statement in Figure 5.6. Again, the clock of the suspension statement is `C1` as for the abortion. Suspension does not force the control flow to jump outside, but just to stay where it is. Hence, when the control flow is at the labels `l2` or `l4` and the condition σ holds, it just stays there. Those control-flow changes can be illustrated in the diagram on the right-hand side by self-loops to the labels. Common for both preemption variants is the behavior when condition σ does not hold. In both cases the normal execution proceeds, as it is e. g. illustrated from label `l2` to `l3`.

In contrast to the delayed strong preemption statements where the preemption can only take place when the control flow is already inside the statement, the immediate variants can also be aborted or suspended when the statement is reached. However, both variants here can be expressed as a macro of the delayed ones (Section 2.2.2), and they are therefore not considered here.

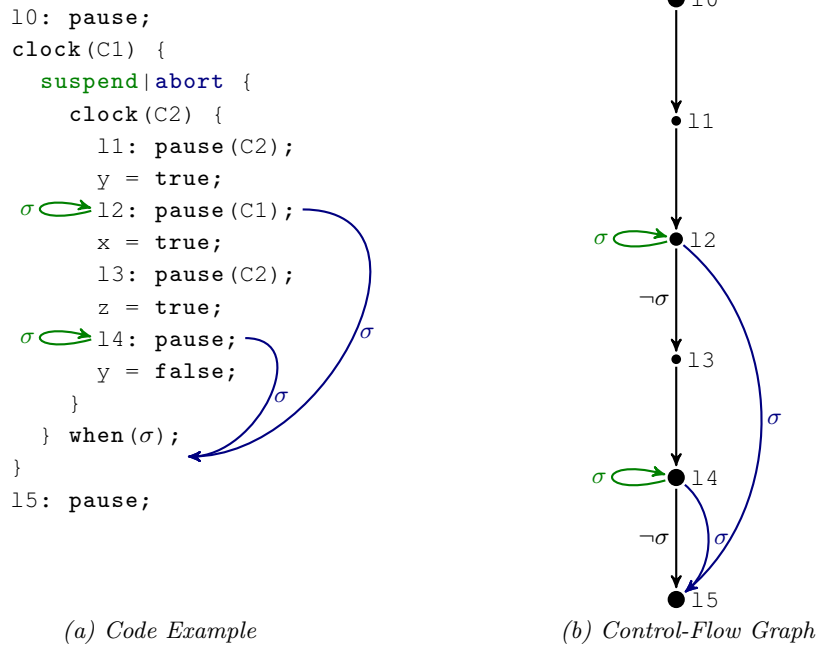


Fig. 5.6. Examples: Preemption with Refined Clocks

5.3.5 Weak Preemption

The weak preemption statements have already proved to be very difficult in the semantics in Chapter 4. The following description illustrates the problems that have to be solved for compiling these statements to the intermediate format. However, the practical relevance of the statements and the complexity that is also introduced here, should justify the decision that the weak preemption statements are not handled by the compile algorithm presented below.

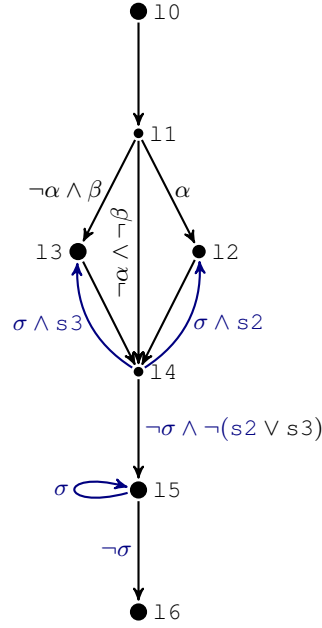
Consider the weak suspension statement and recall that the suspension takes place at the end of a step; Hence, before the next label of the statement clock (or higher) is reached. The control flow must move back from the current label to the label at which the step was started. Consider the example code in Figure 5.7, where the weak versions of the suspension statement are given. For this statement, the suspension can only take place after the first step that entered the statement. This means in the example, just before either the label 15 is reached (this is only possible from label 14), or if the end of the substatement is reached (this is only possible from label 15). However, if the suspension takes place in the last instant starting from label 14, this is only allowed if the first step already entered the substatement (hence the step started from 12 or 13), and the control flow must move back to either 12 or 13. This decision cannot be taken locally without any further information, since when the control flow is at label 14, it is no longer known where it came from. The transition rules in Section 4.3.3 solved this problem by storing the old statement where the step started from. Here, there are two options: (1) store the information by duplicating the control-flow, or (2) store the information by another variable. The second option is illustrated here, and a

```

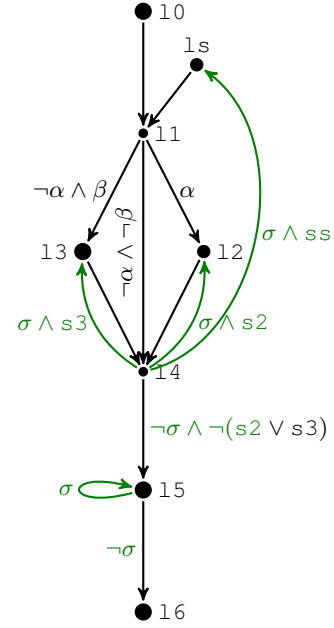
10: pause;
clock(C1) {
  ls: weak
  immediate suspend|
  weak suspend {
    clock(C2) {
      11: pause(C2);
      y = true;
      if(α)
        12: pause(C1);
      if(β)
        13: pause;
      x = true;
      14: pause(C2);
      z = true;
      15: pause;
      y = false;
    }
  } when(σ);
}
16: pause;

```

(a) Code Example



(b) Weak Suspension



(c) Weak Immediate Suspension

Fig. 5.7. Example: Translation of Weak Suspension

new event variable is introduced for each label a step can start from. The variable is then set, when the step starts. In the example, for 12 the event variable s_2 of clock C_1 (the clock of the suspend statement) is introduced, and for 13 respectively. The events are set by additional actions:

$$\begin{aligned}
 C_1 \wedge 12 &\Rightarrow s_2 = \text{true} \\
 C_0 \wedge 13 &\Rightarrow s_3 = \text{true}
 \end{aligned}$$

They are used in the guards of the additional control-flow actions to decide where the control flow has to move. The additional variable is not needed for label 15, since there is just one label where the control flow can move to. Furthermore, for the immediate variant of the suspension statement, the additional label $1s$ is introduced at the beginning, because suspension can take place directly when the statement is entered. Therefore, with this label $1s$ another event ss is also introduced that is set accordingly and used to take the control flow back to the start of the statement. Note that in contrast to the strong counterparts of these statements, the data-flow actions have to be left untouched.

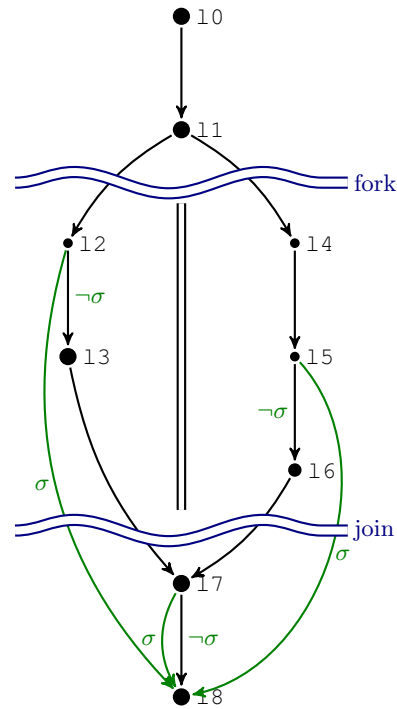
For the weak abortion statement, consider the example in Figure 5.8. For the weak suspension, it seems to be sufficient to just enable the labels from which the step started. The weak abortion, has to abort the step at the end, but it has to ensure that really the end of the step is reached. This is complicated for the parallel statement in combination with the extension. Even if one thread reaches a `pause` statement of the level the abortion should take place, the step can proceed in the other thread (threads synchronize on common `pause`


```

10: pause;
clock(C1) {
  weak abort {
    clock(C2) {
      11: pause;
      {
        12: pause(C2);
        13: pause;
      } || {
        14: pause(C2);
        15: pause(C2);
        16: pause(C1);
      }
      17: pause;
    }
  } when( $\sigma$ );
}
18: pause;

```

(a) Code Example



(b) Weak Abortion

Fig. 5.8. Example: Translation of Weak Abortion

statements). Hence, the thread has to wait, until both threads are in the last instant, and then the abortion can take place. The condition is similar to the one for thread termination, and has to be determined to ensure the correct synchronization when abortion takes place. In the example, when the first thread executes the instant between label 12 and 13, termination cannot take place, since the second thread only moves to label 15. Hence, (weak) abortion can only take place at label 15. However, the illustration shows the general case, where the abortion has also to be *synchronized* by the join of the both threads.

In principle, an implementation of this method is possible, but due to the shown complexity and the lack of practical relevance of the statements, it is not considered in the compile algorithm below.

5.4 Compilation Algorithm

After having pointed out the basic idea of the intermediate format and how some statements are translated to the format, the actual compile algorithm is presented in this section. The compilation procedure for pure QUARTZ is basically separated into two functions, one handles the surface and the other one the depth of a statement. This distinction is an elegant way for solving some issues during compilation like abortion, suspension and also schizophrenia [Schn09]. The algorithm for the extension is changed, but this general structure is kept for some reasons: (1) the algorithm is very complex and therefore it is better to

rely on a version which has been proven correct [ScBS06], (2) even if some issues can be also solved in another way, the distinction makes it more easy, and (3) keeping the same structure makes the differences more clear. However, the original algorithm computes some control-flow predicates which define the interaction of statements, e. g. the term condition identifies the instants in which a statement terminates and is used for the compilation of the sequence of statements: the following statement is started when the previous one terminates. Most of these predicates are computed for each statement, but they are only used at very few places. Furthermore, e. g. `insdS` defines when the control flow is inside of the considered statement and thereby the condition is a disjunction of all labels contained in the statement. The condition is used for the compilation of the parallel statement, as it was explained above, and it is also used for the compilation of the abortion statements in pure QUARTZ, because a statement can be aborted when the control flow is inside of the statement. As explained above, strong abortion for the extension takes place only at several labels and also only if the according clock holds. Hence, the condition needed here is different and depends on the clock. However, a short remark should be given that those different conditions *are fully compatible with the original one and they are equivalent if only one clock is used*.

It is obviously more efficient to do the computation of the predicates in one recursion. However, in order to not complicate the compile algorithm with different predicates and parameters and their complex interaction, they are *computed* when they are needed. The presentation and explanation focuses on translating the behavior of the statement as it was explained above. However, three predicates and parameters are taken from the original algorithm.

- `strts`
The start condition is given as a parameter to the compile function for the surface and identifies the instant when the execution is started. Its purpose can be illustrated very well for the compilation of an assignment: the assignment is translated to a guarded actions with the start condition as guard.
- `inst`
The condition `inst` identifies when a considered statement is instantaneous, hence it is completely executed within one instant. This condition is computed in the compile function for the surface, and it is mainly used to compile a sequence of statements, because thereby a successive statement is started if the preceding one is instantaneous (and it is started).
- `term`
The termination of a statement in an instant is expressed by the condition `term`. It requires that the control flow is inside the statement and it terminates. It is also mainly used to compile a sequence of statements, but it starts the successive statement, when the preceding one terminates.

The original algorithm also provides the abortion and suspension conditions into the recursion and use them at the specific points. The solution that is taken here follows the approach shown above, the compilation of the substatement is done without considering the abort context, and afterwards the obtained guarded actions are modified according to the abort statement. The result is the same, but this approach illustrates the application of the discussion made before. Early versions of the QUARTZ compiler handled the compilation in the same way [ScWe01].

The remaining section is structured as follows. First, some functions are defined which are used later in the compilation of special statements. The functions are e. g. to compute resp. to prepare the special variants of the original *insd* condition. Afterwards, the compilation procedures for surface and depth are explained in detail. The compilation algorithm does not handle the weak preemption statements for the reasons given above.

5.4.1 Definitions

In the way the expressions are composed by the original compile algorithm, each guarded action depends on at least one label having a positive occurrence in its guard (it is assumed that the initial instant is started with the special label *st*). As it was explained for the intermediate format in Section 5.1.1, the label condition for the extension is additionally extended with the clock signal of the *pause* statement the label belongs to. The pair of label and clock identify the instants when execution proceeds from this point and they are considered as control-flow states in the following. Hence, a pair (ℓ, C) where ℓ is a label and C is a clock is called a *state*.

```

function StatesOfExpression(expr)
begin
  switch expr
  case [ $\ell \wedge C$ ]: # expression consists of clock and label
  case [ $C \wedge \ell$ ]:
    return  $\{(\ell, C)\}$ 
  case [ $\text{expr}_1 \wedge \text{expr}_2$ ]: # other conjunction
  case [ $\text{expr}_1 \vee \text{expr}_2$ ]: # disjunction
     $S_1 = \text{StatesOfExpression}(\text{expr}_1)$ 
     $S_2 = \text{StatesOfExpression}(\text{expr}_2)$ 
    return  $S_1 \cup S_2$ 
  case [ $\_$ ]: # every other expression
    return  $\{\}$ 
end

```

Fig. 5.9. Pseudo Code of Function *StatesOfExpression*

The function *StatesOfExpression* given in pseudo code in Figure 5.9 determines exactly those pairs for a given expression. The function recursively decomposes the structure of the expression to find conjunctions of label and clock. For correctness, this function requires that the structure of the Boolean expression is kept as it is constructed. This is assumed for the compile algorithm. Consider the following example:

$$\text{StatesOfExpression}(\ell_0 \wedge C_0 \wedge \alpha \vee \ell_1 \wedge C_1 \wedge \neg \ell_2) \longrightarrow \{(\ell_0, C_0), (\ell_1, C_1)\}$$

The label ℓ_2 is not contained in the result since it has no positive occurrence and there is also no clock. Those sub-expressions can be constructed by the compilation of parallel

```

function StrengthenStateExpr(expr, c, σ)
begin
  switch expr
  case [ℓ ∧ C]: # expression consists of clock and label
  case [C ∧ ℓ]:
    return (if C ≥ c then (ℓ ∧ C) ∧ σ else (ℓ ∧ C))
  case [expr1 ∧ expr2]: # other conjunction
    expr'1 = StrengthenStateExpr(expr1, c, σ)
    expr'2 = StrengthenStateExpr(expr2, c, σ)
    return (expr1 ∧ expr2)
  case [expr1 ∨ expr2]: # disjunction
    expr'1 = StrengthenStateExpr(expr1, c, σ)
    expr'2 = StrengthenStateExpr(expr2, c, σ)
    return (expr1 ∨ expr2)
  case [_]: # every other expression
    return (expr)
end

function StrengthenGuards(A, c, σ)
begin
  return {StrengthenStateExpr(γ, c, σ) ⇒ A | γ ⇒ A ∈ A}
end

```

Fig. 5.10. Pseudo Code of Functions StrengthenStateExpr and StrengthenGuards

statements. Otherwise, labels only occur together with a clock, because the expressions are only constructed in this way by the compile algorithm.

Other functions which are used in the compile algorithm are given in Figure 5.10. The function `StrengthenStateExpr` decomposes a given expression `expr` to the states and strengthens some of them with the given condition σ . The states which are considered have a clock equal or higher to the given clock c . The function's purpose is to e.g. additionally guard the actions which are possibly aborted by adding the abort condition to the states (clock and label) where the abortion takes place. The function is only given in pseudo code but it also requires that the structure of the expressions built by the other functions is kept. It can then traverse the expression's structure to find conjunctions consisting of a clock and a label, because those are the labels the action can be executed from. Note again that at least one of these pairs occur in each guard of each action. The expression σ is then added (as conjunction) to each occurrence of those label-clock pairs having a clock greater or equal to the given one. So, for example the following expressions are changed in the following way:

$$\begin{aligned} \text{StrengthenStateExpr}(\ell_0 \wedge C_0 \wedge \alpha \vee \ell_1 \wedge C_1, C_1, \sigma) &\longrightarrow \ell_0 \wedge C_0 \wedge \alpha \vee \ell_1 \wedge C_1 \wedge \sigma \\ \text{StrengthenStateExpr}(\ell_0 \wedge C_0 \wedge \alpha \vee \ell_1 \wedge C_1, C_0, \sigma) &\longrightarrow \ell_0 \wedge C_0 \wedge \sigma \wedge \alpha \vee \ell_1 \wedge C_1 \wedge \sigma \end{aligned}$$

The results are shown under the assumption that the clock C_1 is smaller than clock C_0 . The exact use of this function is discussed later. The function `StrengthenGuards` also

shown in the figure applies the former function to the guards of a set of guarded actions. In addition to the functions given in pseudo code, the following ones are much simpler and only explained in the following:

- `LabelsOf(\mathcal{S})`
The function returns a set of labels contained in the statement \mathcal{S} . The only statements having labels are **pause** statements and the immediate versions of the suspension statement. All other statements carrying labels are given as macros.
- `StatesOf(\mathcal{S}, C)`
The function returns a set of states contained in the statement \mathcal{S} , but with the restrictions that the clock of the states is at least the given clock, hence:

$$\forall (\ell, c) \in \text{StatesOf}(\mathcal{S}, C) . c \succeq C$$

- `StateCondOf(\mathcal{S}, C)`
The function determines the states in the same way as the former function, but builds a condition out of them:

$$\text{StateCondOf}(\mathcal{S}, C) = \bigvee \{ \ell \wedge C \mid (\ell, C) \in \text{StatesOf}(\mathcal{S}, C) \}$$

Hence, it builds the disjunction of all states with a clock higher or equal to the given clock. These are e. g. exactly the states where an abortion can take place, and the condition is the *extended* version of the *insd* of pure QUARTZ.

With the help of these functions, the compile algorithm can be defined in the following. Note that the results of the functions can be also computed directly in the recursion of the compile algorithm, but to keep it simple, the functions are separated here.

5.4.2 Compile Functions

The actual algorithm follows the approach of the pure QUARTZ and computes the surface and depth related to instants of each statement. The function `CompileSurface` is shown in Figure 5.11, and it expects as inputs the statement's clock c , the start condition `str \mathcal{S}` and the statement \mathcal{S} to compile. The start condition identifies the instant when the statement is started and the statement clock is the lowest visible clock. The function returns the predicate `inst`, a set of guarded actions for the surface and the depth, and a set of reset conditions for local variables. The predicate `inst` expresses when the statement is completely executed in an instant. The compilation of the different statements is explained in the following.

- The immediate and delayed assignments are executed when the statement is started. Hence, the start condition is considered as the guard for the assignments. Since both variants are instantaneously executed, the condition `inst` is **true**. The statement **nothing** is handled in the same way, but obviously, no guarded action is added for it.
- When a **pause** statement is started, its label should hold in the next instant. Therefore, the appropriate guarded action is added to the control flow. The predicate `inst` is obviously **false**, since the instant ends at the statement. Note that the clock of the **pause** statement is not of interest for *reaching* the statement, but it has an impact on leaving the label, which is handled during the compilation of the depth.

```

function CompileSurface(c, strts, S)
begin
  switch S
  case [nothing]:
    return (true, {}, {}, {})
  case [x = τ]: # immediate actions
    return (true, {strts ⇒ x = τ}, {}, {})
  case [next(x) = τ]: # delayed actions
    return (true, {strts ⇒ next(x) = τ}, {}, {})
  case [ℓ: pause(C)]: # pause
    return (false, {}, {strts ⇒ next(ℓ) = true}, {})
  case [if (γ) { S1 } else { S2 }]: # conditional
    (inst1, A1data, A1ctrl, R1) := CompileSurface(c, strts ∧ γ, S1)
    (inst2, A2data, A2ctrl, R2) := CompileSurface(c, strts ∧ ¬γ, S2)
    inst := inst1 ∧ γ ∨ inst2 ∧ ¬γ ∨ inst1 ∧ inst2
    return (inst, A1data ∪ A2data, A1ctrl ∪ A2ctrl, R1 ∪ R2)
  case [S1; S2]: # sequence
    (inst1, A1data, A1ctrl, R1) := CompileSurface(c, strts, S1)
    (inst2, A2data, A2ctrl, R2) := CompileSurface(c, strts ∧ inst1, S2)
    return (inst1 ∧ inst2, A1data ∪ A2data, A1ctrl ∪ A2ctrl, R1 ∪ R2)
  case [S1 || S2]: # parallel threads
    (inst1, A1data, A1ctrl, R1) := CompileSurface(c, strts, S1)
    (inst2, A2data, A2ctrl, R2) := CompileSurface(c, strts, S2)
    return (inst1 ∧ inst2, A1data ∪ A2data, A1ctrl ∪ A2ctrl, R1 ∪ R2)
  case [do { S' } while(γ)]:
    return CompileSurface(c, strts, S')
  case [abort { S' } when(γ)]:
    return CompileSurface(c, strts, S')
  case [suspend { S' } when(γ)]:
    return CompileSurface(c, strts, S')
  case [clock (C) { S' }]: # clock declaration
    return CompileSurface(C, strts, S')
  case [{α x; S'}]: # variable declaration
    S = StatesOfExpression(strts)
    L = LabelsOf(S')
    if ∃(c', ℓ) ∈ S. (ℓ ∈ L) ∨ ¬(c ≥ c') then Error()
    (inst, Adata, Actrl, R) = CompileSurface(C, strts, S')
    R' := R ∪ {(x, strts)}
    return (inst, Adata, Actrl, R')
end

```

Fig. 5.11. Pseudo Code of Function CompileSurface

- The `if` statement strengthens the start condition for the branches and combines the guarded actions obtained from the compilation of both. The condition `inst` is also combined in the same way as for pure QUARTZ.
- The sequence of statements is compiled as follows: when the whole sequence is started, then the first statement is, hence, it has the same start condition. The second part of the sequence is started in the same instant only if the first one is instantaneous (and the whole sequence is started). The resulting guarded actions are combined, and the sequence can only be instantaneous, when each part of the sequence is so.
- When the parallel statement is started, also both threads are started. Hence, they also get the same start condition. The resulting guarded actions are combined, and the parallel execution is only instantaneous, when both threads are so.
- The abortion statement influences the execution only when the control flow is inside and not when the statement is started. Therefore, the result for the whole statement is the result obtained from the substatement.
- Like for abortion, the suspension also does not influence the first instant (even not the first step). Therefore, the result for the whole statement is the result obtained from the substatement.
- A clock declaration only defines a new clock for a substatement, and therefore, the clock for compiling the substatement is updated. It does not have any other impact on the execution and therefore it simply returns the result obtained from the substatement.
- For entering the scope of a variable, it has to be ensured that the variable is reset when it is needed. The start condition is added as reset condition for this variable. However, the intermediate format has the restriction that the reset condition only holds in an instant of the clock of the variable. Furthermore, to ensure that the scope of the same variable is not entered in the same step, it ensures that the states occurring in the start condition are not contained in the scope. If this is the case, the compilation can simply forward the result from the substatement.

The function `CompileDepth` is shown in Figure 5.12. It expects as inputs the statement's clock c and the statement to compile \mathcal{S} . The function returns the predicate `term`, a set of guarded actions for surface and depth, and a set of reset conditions for local variables. The predicate `term` expresses when the statement terminates in an instant also requiring that the control flow is inside the statement, hence at least one label of the statement holds. The compilation of the various statements is explained in the following.

- The immediate and delayed assignments are executed when the statement is started as it is handled in the surface. Since they do not have any behavior in the depth, they have a simple result. The statement `nothing` is handled in the same way.
- The `pause` statement of clock C is left when its label and its clock holds. Therefore, the term condition is composed of both. There are no actions to execute.
- The conditional `if` statement simply combines the results obtained from both branches. Note that starting the statement is handled by the surface. The statement terminates when one of its branches terminate.
- The compilation of the depth of the sequence $\mathcal{S}_1; \mathcal{S}_2$ requires to use the compilation of the surface. Thereby, the depth is composed of the depth of \mathcal{S}_1 , the surface of \mathcal{S}_2 , and the depth of \mathcal{S}_2 . The start condition for the surface of the second statement is the termination

condition of the first one. The results are then simply combined. The whole statement terminates when either the first one terminates and the second one is instantaneous, or when just the second one terminates.

- For the compilation of the depth of the parallel statement, simply both threads have to be considered. Like for the conditional `if` statement, the results can be simply combined, since the depth presumes that the statement was already started. However, the termination needs more effort, since the threads need to synchronize their termination. As explained above, the termination condition is composed in the same way as for pure QUARTZ, hence, either both threads terminate in the same instant, or one terminates and the other one already had terminated.
- The issues for compiling the abort statement have been discussed above, the strong abortion is possible at each label with a clock higher or equal to the statement clock. The abortion is covered by the termination condition: in case the control flow is at a label with a clock higher or equal to the statement clock, and the condition holds, the statement terminates. Additionally, the execution inside the substatement must be stopped, and therefore, the guards of the actions having the right label and clock combination are strengthened with the negation of the abortion condition.
- The depth of the strong suspension behaves similar to the abortion statement: the original behavior has to be prevented at labels of the statement's clock or a higher clock. Therefore, the guards are strengthened in the same way. However, the execution is not aborted, but only stopped. Therefore, additional guarded actions are needed which keep the control flow at the labels where it currently is, in case of suspension. Furthermore, the termination condition needs to be adjusted, but only for the labels where also abortion could take place. Hence, it is strengthened in the same way as the guards.
- Like for the compilation of the surface, the clock declaration statement simply updates the statement clock for compiling the depth of the substatement.
- The declaration of local variables is mostly handled by the surface; there is nothing to do for the depth. Hence, the result from the substatement is the result of the whole statement.

Finally, the result of the compilation is then obtained by the function `Compile` shown in Figure 5.13 by composing the sequence of surface and depth of the whole program. Thereby, the surface is started when the whole program is started which is expressed by the condition $st \wedge C0$. The result only consists of the guarded actions and reset conditions obtained from both functions, and the additional computed predicates are only used during compilation.

5.5 Checking Constructive Abstractions

In Section 4.5, an interpreter based on the SOS rules was given to define the semantics of programs. Thereby, the constructivity for each instant was, like for pure QUARTZ, defined by the reaction rules, but the constructivity *across* several instants depends on the choice of the clocks and on the execution of the default reaction. Two abstractions were given ensuring that a correct execution can be scheduling independent. Furthermore, checking whether a program is constructive or not is hard [ScBr08]. Therefore, a common solution is to use abstractions that can be easily and fast checked to determine if a program is constructive. For


```

function CompileDepth( $c, S$ )
begin
  switch  $S$ 
  case [nothing]:
  case [ $x = \tau$ ]: # immediate actions
  case [next( $x$ ) =  $\tau$ ]: # delayed actions
    return (false, {}, {})
  case [ $\ell$ : pause( $C$ )]: # pause
    return ( $\ell \wedge C, \{\}, \{\}) \#\{\ell \wedge C \wedge \text{susp}_S(C) \Rightarrow \text{next}(\ell) = \text{true}\}$ )
  case [if ( $\gamma$ ) {  $S_1$  } else {  $S_2$  }]: # conditional
    (term1,  $A_1^{\text{data}}$ ,  $A_1^{\text{ctrl}}$ ) := CompileDepth( $c, S_1$ )
    (term2,  $A_2^{\text{data}}$ ,  $A_2^{\text{ctrl}}$ ) := CompileDepth( $c, S_2$ )
    return (term1  $\vee$  term2,  $A_1^{\text{data}} \cup A_2^{\text{data}}$ ,  $A_1^{\text{ctrl}} \cup A_2^{\text{ctrl}}$ )
  case [ $S_1; S_2$ ]: # sequence
    (term1,  $A_1^{\text{data}}$ ,  $A_1^{\text{ctrl}}$ ,  $\mathcal{R}_1$ ) := CompileDepth( $c, S_1$ )
    (inst2,  $A_2^{\text{data}}$ ,  $A_2^{\text{ctrl}}$ ,  $\mathcal{R}_2$ ) := CompileSurface( $c, \text{term}_1, S_2$ )
    (term2,  $A_3^{\text{data}}$ ,  $A_3^{\text{ctrl}}$ ,  $\mathcal{R}_3$ ) := CompileDepth( $c, S_2$ )
    term := term1  $\wedge$  inst2  $\vee$  term2
    return (term,  $A_1^{\text{data}} \cup A_2^{\text{data}} \cup A_3^{\text{data}}$ ,  $A_1^{\text{ctrl}} \cup A_2^{\text{ctrl}} \cup A_3^{\text{ctrl}}$ ,  $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$ )
  case [ $S_1 \parallel S_2$ ]: # parallel threads
    (term1,  $A_1^{\text{data}}$ ,  $A_1^{\text{ctrl}}$ ,  $\mathcal{R}_1$ ) := CompileDepth( $c, S_1$ )
    (term2,  $A_2^{\text{data}}$ ,  $A_2^{\text{ctrl}}$ ,  $\mathcal{R}_2$ ) := CompileDepth( $c, S_2$ )
    insd1 :=  $\bigvee$  LabelsOf( $S_1$ )
    insd2 :=  $\bigvee$  LabelsOf( $S_2$ )
    term := term1  $\wedge$  term2  $\vee$  term1  $\wedge$   $\neg$ insd2  $\vee$  term2  $\wedge$   $\neg$ insd1
    return (term,  $A_1^{\text{data}} \cup A_2^{\text{data}}$ ,  $A_1^{\text{ctrl}} \cup A_2^{\text{ctrl}}$ ,  $\mathcal{R}_1 \cup \mathcal{R}_2$ )
  case [abort {  $S'$  } when( $\sigma$ )]:
    (term,  $A^{\text{data}}$ ,  $A^{\text{ctrl}}$ ,  $\mathcal{R}$ ) := CompileDepth( $c, S'$ )
    insd $c$  :=  $\bigvee\{\ell \wedge c \mid (\ell, c) \in \text{StatesOf}(S', c) \wedge C \succeq c\}$ 
     $A^{\text{data}}$  := StrGuards( $A^{\text{data}}$ ,  $c, \neg\sigma$ )
     $A^{\text{ctrl}}$  := StrGuards( $A^{\text{ctrl}}$ ,  $c, \neg\sigma$ )
    term := StrengthenStateExpr(term,  $c, \sigma$ )  $\vee$  insd $c$   $\wedge$   $\sigma$ 
    return (term,  $A^{\text{data}}$ ,  $A^{\text{ctrl}}$ ,  $\mathcal{R}$ )
  case [suspend {  $S'$  } when( $\sigma$ )]:
    (term,  $A^{\text{data}}$ ,  $A^{\text{ctrl}}$ ,  $\mathcal{R}$ ) := CompileDepth( $c, S'$ )
     $A_S^{\text{ctrl}}$  :=  $\{\ell \wedge C \wedge \sigma \Rightarrow \text{next}(\ell) = \text{true} \mid (\ell, C) \in \text{StatesOf}(S', c) \wedge C \succeq c\}$ 
     $A^{\text{data}}$  := StrGuards( $A^{\text{data}}$ ,  $c, \neg\sigma$ )
     $A^{\text{ctrl}}$  := StrGuards( $A^{\text{ctrl}}$ ,  $c, \neg\sigma$ )
    term' := StrengthenStateExpr(term,  $c, \sigma$ )
    return (term',  $A^{\text{data}}$ ,  $A^{\text{ctrl}} \cup A_S^{\text{ctrl}}$ ,  $\mathcal{R}$ )
  case [clock ( $C$ ) {  $S'$  }]: return CompileDepth( $C, S'$ )
  case [{ $\alpha$  x;  $S'$  }]: return CompileDepth( $c, S'$ )
end

```

Fig. 5.12. Pseudo Code of Function CompileDepth

```

function Compile( $\mathcal{S}$ )
begin
  (inst,  $\mathcal{A}_1^{\text{data}}$ ,  $\mathcal{A}_1^{\text{ctrl}}$ ,  $\mathcal{R}_1$ ) := CompileSurface( $C_0$ ,  $\text{st} \wedge C_0$ ,  $\mathcal{S}$ )
  (term,  $\mathcal{A}_2^{\text{data}}$ ,  $\mathcal{A}_2^{\text{ctrl}}$ ,  $\mathcal{R}_2$ ) := CompileDepth( $C_0$ ,  $\mathcal{S}$ )
  return ( $\mathcal{A}_1^{\text{data}} \cup \mathcal{A}_2^{\text{data}}$ ,  $\mathcal{A}_1^{\text{ctrl}} \cup \mathcal{A}_2^{\text{ctrl}}$ ,  $\mathcal{R}_1 \cup \mathcal{R}_2$ )
end

```

Fig. 5.13. Pseudo Code of Function Compile

example, as already pointed out in Section 2.2.7, for the synthesis of QUARTZ to sequential software, the guarded actions must be ordered in a sequence to be scheduled correctly. Also for hardware synthesis, even if the hardware circuit generated for constructive programs will stabilize, most backend tools for generation tools cannot deal with cycles. Therefore, checking for acyclic dependencies is a common approach to ensure that a program is constructive.

However, the constructivity in an instant as it is defined by the reaction rules can be handled in the same way as for pure QUARTZ. Here, it is focused on the dependencies between instants of different or also of the same clock based on the abstractions defined with the interpreter. Thereby, it must be ensured that a variable is not written after it was read.

A fast check can therefore be defined based on the guarded actions obtained from the compiler. As explained above, the whole program can be represented by a control-flow graph. At this graph it is needed to be checked that for each variable written in an instant of a lower clock it is not read before that instant.

5.6 Summary

The compilation to an intermediate format abstracting from issues present in the source language proved to be useful to simplify the synthesis tools: the effort for handling all aspects of the source language has only to be done once. The same approach is kept for the extension, but the original intermediate format could obviously not be kept, since it does not handle multiple clocks. The extension of the intermediate format was motivated in this chapter, before the translation of several statements to this format has been discussed. Finally, the whole compilation algorithm was presented, which is quite similar to the algorithm for pure QUARTZ, except for some special cases which were explained. Especially for a common problem of synchronous languages, namely schizophrenia, a solution for the compilation was proposed that handles simple cases very efficiently.

Chapter 6

Hardware Synthesis

The synthesis of the original intermediate format AIF to synchronous hardware circuits was presented in Section 2.2.7. Each macro step of the original QUARTZ program was translated to one clock cycle of the hardware clock. Communication of inputs and outputs between the module and the environment happens in each clock cycle. Even though this is not the only possible way to generate hardware, it is nevertheless an obvious representation relying on the fact that a finite number of micro steps are executed in a macro step. This changes when the extension is considered, since now an arbitrary number of substeps can be executed within a step of the module's clock. The following two options come to mind for translating the extended intermediate format to synchronous hardware:

- The first option is to follow the original QUARTZ approach and translate *each step* of the module clock to one hardware clock cycle. This translation would keep the same interface as the original translation, and communication between the module and its environment happens in each clock cycle. However, since a step consists of several substeps, those must be all executed within this single cycle. A costly analysis is needed to *unroll* potential loops and to *chain* explicitly all states reachable during the execution of one macro step. Nevertheless, concurrent execution of independent clocks could be perfectly parallelized.
- The second option is to translate *each instant* of the system to one hardware clock cycle. Then, the execution of a step of the module needs several hardware clock cycles and the interface to the environment must be changed: the environment must know when a step is finished to get the output values and provide new input values. Independent execution of unrelated clocks can be represented by clock inputs triggering the execution of the instants and following the restrictions given by the control flow of the program. The clock inputs can be set by a *scheduler* to trigger the computation.

The first option needs an expensive analysis for unrolling the steps and determining all possible reachable states within a macro step. The critical path would be increased and the hardware-clock frequency will be reduced. This first option is mentioned to show that there are different solutions, but the second choice is taken in this work, because it provides a straightforward translation of instants to hardware clock cycles without needing further analysis. However, the real execution time of a step of the module depends on the number of substeps and will be dynamic, but the instants are much smaller and the frequency can be increased. Additionally, to elaborate different possibly execution schemes, the hardware

circuit is constructed of two parts. The first one represents the *functional part* describing the actual behavior of the program with the independence of unrelated clocks. The second part represents the *scheduler* triggering the actual computation by activating the clocks for execution of an instant. The constructivity of programs that was defined in Chapter 4 is scheduling independent, hence, each scheduler fulfilling some basic requirements lead to a correct execution of the system. An advantage of the separation is that if another notion of constructivity would be considered, the scheduler needs to be changed, but the functional part of the translation can be the same.

Note also that the presented translation is based on constructive programs, hence, a behavior of the program that cannot be constructively determined cannot be computed by the system obtained by this translation. Furthermore, also programs with write conflicts (these are also not constructive), are not represented correctly, since the result of write conflicts is undefined. Hence, the starting point for the code generation are programs that are constructive and free of write conflicts.

The remaining chapter is structured as follows. First, the overall structure of the generated hardware with separation into a functional part and a scheduler is described. Then, the actual translation of the functional part is explained, and finally, the synthesis of one possible scheduling independent scheduler is explained.

6.1 Overall Structure

As introduced above, the goal is to translate programs from the extended intermediate format to synchronous hardware where in each hardware clock cycle one instant of the original program is executed. Independent execution of unrelated clocks is modeled by an additional scheduler which can take one possible clock choice to execute the next instant, however, since the instants of different clocks are also unrelated, the scheduler can also select more than one clock to execute the independent instants at once. The clear distinction allows to separate the functional behavior from the chosen execution.

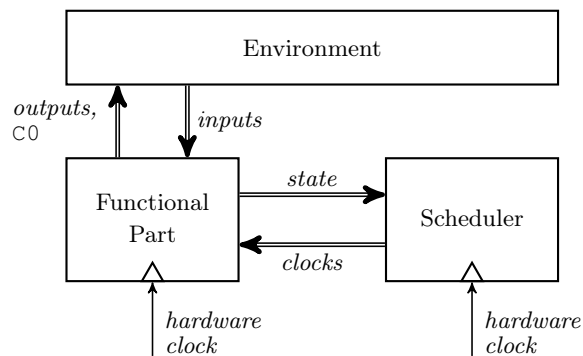


Fig. 6.1. Overall Structure of Generated Hardware

The structure of the generated hardware is illustrated by Figure 6.1. Both, the functional part and the scheduler, are driven by a dedicated hardware clock. Note that the clocks

defined in QUARTZ are used to define substeps in the description language, and that these clocks are not equal to the hardware clock that is only used to trigger the execution of the instants. The environment sets the input values for the functional part and reads the output values from it, but the environment needs to be notified about the start of a new step to provide new input values. Therefore, the signal `C0` representing the module clock is provided in addition to the other outputs to the environment. The scheduler triggers the computation by selecting one or more (QUARTZ) clocks based on the given state of the system. Thereby, the scheduler has to select the clocks according to the interpreter (defining the semantics): virtually, the set of clocks from the transition rules has to be determined and some of the smallest ones have to be chosen. The signals representing the state of the system can vary depending on the complexity of the scheduler, therefore, it is not further specified here. The scheduler presented below will e. g. only use the labels of the systems.

6.2 Functional Part

This section explains the translation of the functional part to hardware. It first recalls the hardware representation introduced in Section 2.2.7 which is an equation system being evaluated in each clock cycle. Afterwards, a general representation of control flow and data flow by equation systems is given. In the case of data flow, this representation can be optimized if certain conditions hold for the intermediate format. The conditions are introduced and the optimizations are explained based on the most general representation. Finally, the optimizations are summarized and their relation to the original hardware synthesis are highlighted.

6.2.1 Representation of Hardware

Synchronous hardware circuits can be described by equation systems which, given a library for all operators in QUARTZ, can be syntactically translated to hardware description languages such as VERILOG or VHDL. The equation system is represented by three different kinds of equations. The first type represents *wires* which are directly connected to logic gates so that the computed value is immediately available in the same clock cycle. Such an immediate equation has the form

$$x = \tau$$

for a variable x set to the current value of the expression τ . State elements such as *registers* are represented by the remaining equations. Each register is defined by two equations, one for the initial clock cycle and one for subsequent transitions:

$$\begin{aligned} \text{init}(x) &= \tau_1 \\ \text{next}(x) &= \tau_2 \end{aligned}$$

The registers are updated by a clock transition of the hardware clock with the value of the expression τ_2 evaluated in the last clock cycle, and $\text{init}(x)$ defines a value for the first clock cycle. Note that the expression $\text{next}(x)$ does not correspond to a delayed assignment in extended QUARTZ, because the register is updated in the next clock cycle of the hardware clock, and not with the beginning of a new step of a clock defined in QUARTZ.

6.2.2 Translation of Control Flow

Like for pure QUARTZ, the translation of the control flow to equation systems is done for each label separately, but setting and resetting the label is different. A label ℓ is written by some delayed guarded actions (note that the control flow does not contain immediate actions):

$$\begin{aligned}\gamma_1 &\Rightarrow \text{next}(\ell) = \text{true} \\ \gamma_2 &\Rightarrow \text{next}(\ell) = \text{true} \\ &\dots \\ \gamma_n &\Rightarrow \text{next}(\ell) = \text{true}\end{aligned}$$

All labels are set to `false` for the initial instant, except the special start label `st` separately considered below. When a label is activated by a delayed assignment, this assignment does not depend on the clock of the label but sets the label active for the next instant. The behavior is different to delayed assignments to normal variables in extended QUARTZ, but it can be easily translated to registers that are set for the next clock cycle. The label ℓ is only deactivated if an instant of the clock of ℓ is executed (cf. `await` semantics). This leads to the following representation of the label by a register:

$$\begin{aligned}\text{init}(\ell) &= \text{false} \\ \text{next}(\ell) &= \underbrace{\gamma_1 \vee \gamma_2 \vee \dots \vee \gamma_n}_{\text{guards}} \vee \underbrace{(\ell \wedge \neg \text{clock}(\ell))}_{\text{default}}\end{aligned}$$

The first part of the expression to define the next value of the label is constructed by the guards of the guarded actions, because the value will be set to `true` when at least one guard is evaluated to `true`. The second part of the expression defines the default value taken when no delayed guarded action sets a value to ensure that the label is kept active as long as its clock does not hold: when an instant of the label's clock is executed, the label will be reset for the next instant, if it is not activated again by a delayed guarded action. There are no guarded actions defined for the special start label `st` and therefore it is translated to a register as follows:

$$\begin{aligned}\text{init}(\text{st}) &= \text{true} \\ \text{next}(\text{st}) &= \text{st} \wedge \neg \text{clock}(\text{st})\end{aligned}$$

Whereas `clock(st)` is the module's clock `C0` and the start label is set from the beginning and it is reset after the first instant is executed.

6.2.3 Translation of Data Flow

The translation of the data flow is more sophisticated and the general case is presented in this section. In contrast to control-flow label, variables can be written by immediate *and* delayed assignments. Thereby, the delayed assignments assign a value for the next step of the variable's clock and not, like for labels, for the next executed instant. The value of the delayed assignments must be stored until the following step starts. Furthermore, reset conditions defining when a variable needs to be reset must be handled. In the general case, three registers are needed for each variable: one carries the value of the current step, and the other collect and store the value of a delayed assignment until the next step of the

variable's clock begins. However, three registers for one variable in QUARTZ sounds like an overhead, but the general case is described first, before some special cases are introduced with optimizations where the number of registers can be reduced. Like for the control flow, each variable can be translated separately. For the translation, assume a variable x that is written by the following guarded actions:

$$\begin{array}{ll}
 \gamma_1^i \Rightarrow x = \tau_1^i & \gamma_1^d \Rightarrow \text{next}(x) = \tau_1^d \\
 \gamma_2^i \Rightarrow x = \tau_2^i & \gamma_2^d \Rightarrow \text{next}(x) = \tau_2^d \\
 \dots & \dots \\
 \gamma_n^i \Rightarrow x = \tau_n^i & \gamma_m^d \Rightarrow \text{next}(x) = \tau_m^d
 \end{array}$$

For the variable x , one wire also having the name x , and the three registers with the following names are used:

- x^{next}
 The register stores the value of a delayed assignment executed during a step of the clock of x (note that due to the absence write conflicts, there can be at most one delayed assignment setting a value for the next step) and carries the value until the next step starts.
- x^{nas}
 This Boolean register stores the information whether a delayed assignment was executed in the last step or not. It is required to determine at the beginning of a new step if the value of x^{next} must be assigned or not.
- x^{prv}
 A step of the clock of x can consist of instants of a smaller clock and the value of x must be kept for the instants. Therefore, the register x^{prv} stores the value of x from the previous instant.

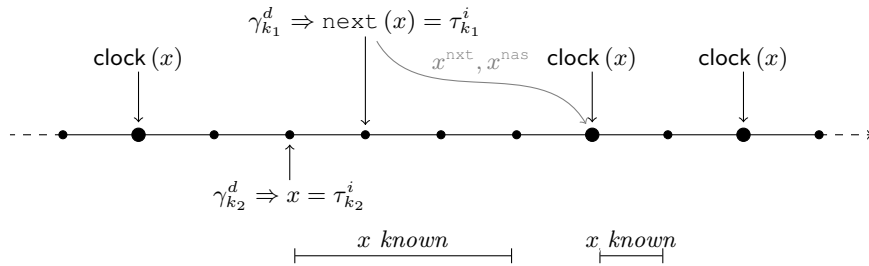


Fig. 6.2. Execution Trace for x

To explain the idea of the translation, consider the illustration of an execution trace for the variable x in Figure 6.2. Each dot on the timeline represents an instant (a clock cycle of the hardware clock). The instants are distinguished by their clocks, where the big dots are instants of $\text{clock}(x)$ (or higher) and the small ones are instants of a lower or unrelated clock. The distinction is made in this way, because smaller instants are micro steps from the view of x and greater ones are the instants a step of $\text{clock}(x)$ starts (or ends). Remember, a step of $\text{clock}(x)$ starts with an instant of $\text{clock}(x)$ and ends before the next instant of $\text{clock}(x)$.

During this step, the variable x has logically one value which may not be known from the beginning, but can only be used as soon as it was assigned. Hence, the variable can get its value from an immediate assignment in an instant and will then be *known* for the remaining step and the register x^{prv} is used to store this value. A delayed assignment that takes places during a step is assigned to x^{next} (and x^{nas} is set to true), and carried until the next step starts (instant of $\text{clock}(x)$) where x^{nas} is used to decide whether the value of x^{next} is used or not.

The semantics extended the values a variable can hold with the symbol \perp to show that the value for the current step has not been determined yet. This value is not explicitly modeled in the synthesis, but since the translation relies on constructive programs, it is ensured that a variable is not written after it was read. Therefore, the *default reaction* can be executed in the first instant of a step independent of an assignment would overwrite the value again. Hence, the value is either changed by an assignment, or the default value is used. Finally, the guarded actions for the variable x are translated to the following equations:

$$\begin{aligned}
& \text{init}(x^{\text{next}}) = \text{default}(x) \\
& \text{next}(x^{\text{next}}) = \begin{cases} \tau_1^d & : & \gamma_1^d \\ \tau_2^d & : & \gamma_2^d \\ \vdots & : & \vdots \\ \tau_m^d & : & \gamma_m^d \\ \text{trani}(x) & : & \text{clock}(x) \\ x^{\text{next}} & : & \text{default} \end{cases} \\
& \text{init}(x^{\text{nas}}) = \text{false} \\
& \text{next}(x^{\text{nas}}) = \begin{cases} \gamma_1^d \vee \gamma_2^d \vee \dots \vee \gamma_m^d \\ (x^{\text{nas}} \wedge \neg \text{clock}(x)) \end{cases} \\
& \text{init}(x^{\text{prv}}) = \text{default}(x) \\
& \text{next}(x^{\text{prv}}) = x
\end{aligned}
\quad x = \begin{cases} \tau_1^i & : & \gamma_1^i \\ \tau_2^i & : & \gamma_2^i \\ \vdots & : & \vdots \\ \tau_m^i & : & \gamma_m^i \end{cases} \quad (a)
\quad \begin{cases} \text{default}(x) & : & \text{reset}(x) & (b) \\ x^{\text{next}} & : & \text{clock}(x) \wedge x^{\text{nas}} & (c) \\ \text{trans}(x) & : & \text{clock}(x) & (d) \\ x^{\text{prv}} & : & \text{default} & (e) \end{cases} \quad (\text{E}_1)$$

Register Definitions

Wire Definition

The register x^{prv} is defined by Equation E₁ to have the value of x from the previous clock cycle and is initialized with the default value for x . The register x^{next} and x^{nas} stores value and occurrence of delayed assignments during a step: if a delayed assignment is executed in a step, x^{next} gets its value and keeps it until the next step starts. The register x^{nas} is a Boolean flag and just stores whether a delayed assignment occurred or not. Therefore, it is set to true when one of the guards hold and the value is kept until the next instant of $\text{clock}(x)$. The translation for this register is similar to the translation of labels.

As covered in the definition by the cases (a), the value of x is determined by an immediate action when its guard holds. The next Case (b) which can be taken resets the variable x to its default value when the reset condition holds, i. e. its scope is entered in the source code. This happens before the delayed assignments are checked, because delayed assignments executed in the previous step cannot refer to the variable in the new scope. However, delayed assignments from the previous step are assigned to x when a new step is started and a delayed

assignment occurred as covered by Case (c). Case (d) takes care of the default reaction which is executed at the beginning of a new step: events are reset to $\text{default}(x)$ and memorized variables keep the last value. In this way, it is ensured that x has the default value for the whole step when no assignment will change it. Finally, the Case (e) just keeps the value from the last instant which is only activated in instants of smaller or unrelated clocks (the small dots in the picture). The expression $\text{trans}(x)$ executes the default reaction depending on the storage type of the variable x :

$$\text{trans}(x) := \begin{cases} \text{default}(x) & : x \text{ is event variable} \\ x^{\text{prv}} & : x \text{ is memorized variable} \end{cases}$$

Event variables are reset to their default value and memorized variables store the value from the previous step. The case where $\text{clock}(x)$ holds is used in the definition of x^{next} , but was not explained so far. Currently, the value of x^{next} is only used when x^{nas} holds, and this is reset when $\text{clock}(x)$ holds, hence, the value assigned in this case is never used. However, it is defined as $\text{trani}(x)$ with:

$$\text{trani}(x) := \begin{cases} \text{default}(x) & : x \text{ is event variable} \\ x & : x \text{ is memorized variable} \end{cases}$$

The value of x^{next} is *initialized* for a new step with the value of x in the first instant (for memorized variables), or the default value of x (for event variables). This definition will allow some simplifications in the following, but it does not make any difference in this general case.

6.2.4 Optimizations for Data-Flow

This section discusses simplifications of the generated equations based on special cases of the intermediate format. According to the separate translation of each variable, these optimizations can be also done individually. Optimizations for pure QUARTZ can e. g. be done, when an event variable is not written by delayed assignments, then the register can be omitted. The extension provides more capability for simplification based on the instants in which variables are read and written. Some of them are discussed in the following by defining a certain property of a variable x first, and simplifying the above general translation afterwards. Finally, all cases are compared.

Events

The general representation of the data flow for a variable has been given above for both, event variables and memorized variables. Even though this is a good starting point for both, it is now simplified for event variables. Thereby, event variables are reset to their default value when a new step of their clock is started, whereas memorized variables are set to their value of the last step. The definition given above also sets the register storing the values of delayed assignment to the default value for events. Hence, if there are no delayed assignments executed in a step, the register x^{next} still has the default value and for defining x , it has not to be distinguished whether to set it to the default value or to x^{next} .

For a more precise discussion, consider Figure 6.3 illustrating an execution trace from the view of the event variable x . The considered step starts in the instant of $\text{clock}(x)$ at the point

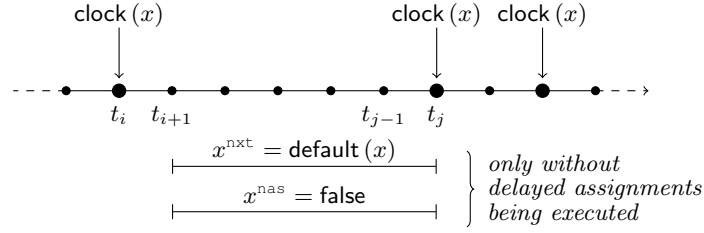


Fig. 6.3. Execution Trace for x (Event)

of time t_i and ends before the next instant of $\text{clock}(x)$ at the point of time t_j . The value of x in the instant t_k is referred to by x_k in the following to distinguish the different values. Assume that there is no delayed assignment fired in the considered step and $\gamma_1^d, \gamma_2^d, \dots, \gamma_m^d$ are evaluated to **false** in the instants $t_i, t_{i+1}, \dots, t_{j-1}$, then the value of x^{next} behaves as follows:

$$\text{default}(x) = x_{i+1}^{\text{next}} = x_{i+2}^{\text{next}} = \dots = x_j^{\text{next}}.$$

Hence, it keeps constantly the default value of the event variable x in this step. However, the same holds for x^{nas} being constantly **false** for those instants, because the assumption is that no delayed assignment is executed. The crucial point here is that $x_j^{\text{next}} = \text{default}(x)$ and that $x_j^{\text{nas}} = \text{false}$. Considering the original definition of Equation E₁, the following cases can be rewritten as

$$\begin{array}{ccc} \vdots & & \vdots \\ x^{\text{next}} & : \text{clock}(x) \wedge x^{\text{nas}} & \Rightarrow & x^{\text{next}} & : \text{clock}(x) \wedge x^{\text{nas}} \\ \text{trans}(x) & : \text{clock}(x) & & \text{trans}(x) & : \text{clock}(x) \wedge \neg x^{\text{nas}} \\ \vdots & & & \vdots & \end{array}$$

making more clear that the second case is only reached when x^{nas} is **false** (in an instant of $\text{clock}(x)$). But in this case x^{next} is equal to $\text{default}(x)$ and both cases can be combined:

$$\begin{array}{ccc} \vdots & & \vdots \\ x^{\text{next}} & : \text{clock}(x) \wedge x^{\text{nas}} & \Rightarrow & x^{\text{next}} & : \text{clock}(x) \\ \text{trans}(x) & : \text{clock}(x) \wedge \neg x^{\text{nas}} & & & \\ \vdots & & & \vdots & \end{array}$$

As a result, the register x^{nas} is no longer used and can be completely omitted and the following complete definition is obtained:

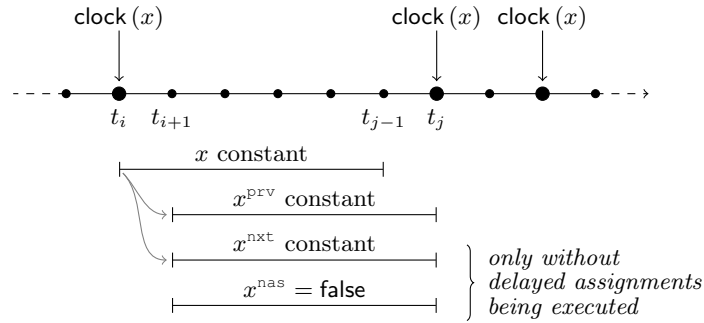
$$\begin{aligned}
& \text{init}(x^{\text{next}}) = \text{default}(x) \\
& \text{next}(x^{\text{next}}) = \begin{cases} \tau_1^d & : & \gamma_1^d \\ \tau_2^d & : & \gamma_2^d \\ \vdots & & \vdots \\ \tau_m^d & : & \gamma_m^d \\ \text{trani}(x) & : & \text{clock}(x) \\ x^{\text{next}} & : & \text{default} \end{cases} \\
& \text{init}(x^{\text{prv}}) = \text{default}(x) \\
& \text{next}(x^{\text{prv}}) = x
\end{aligned}
\qquad
x = \begin{cases} \tau_1^i & : & \gamma_1^i \\ \tau_2^i & : & \gamma_2^i \\ \vdots & & \vdots \\ \tau_m^i & : & \gamma_n^i \\ \text{default}(x) & : & \text{reset}(x) \\ x^{\text{next}} & : & \text{clock}(x) \\ x^{\text{prv}} & : & \text{default} \end{cases} \quad (\text{E}_2)$$

Register Definitions

Wire Definition

Variable x only used in Instants of $\text{clock}(x)$

The reason for having three registers for one memorized variable x comes from the fact that the value of x must be available in the smaller instants during a step of $\text{clock}(x)$. However, this is e. g. not required for variables defined on one of the smallest clocks (leaves of the clock tree), since there are no smaller instants which could use (reading or writing) the variable.

**Fig. 6.4.** Execution Trace for x (Only Read in Instants $\text{clock}(x)$)

Consider first the assumption that the variable x is only written in instances of clock of x . Since this assumption is automatically fulfilled by delayed assignments setting the value in the first instant of the next step of the clock of x , the real requirement is only given for immediate assignments:

$$\text{write}^i(x) \implies \text{clock}(x). \quad (\text{P}_1)$$

Note that the property can be easily checked by the states occurring in all guarded actions that immediately assign the variable x . To elaborate the effects of this property, consider the illustration in Figure 6.4 showing an execution from the view of variable x with the two points of time t_i and t_j representing instances of $\text{clock}(x)$ (with $i < j$). The instances in between are of a lower or unrelated clock such that the considered step of $\text{clock}(x)$ consists of the instances t_i, \dots, t_{j-1} . Due to the above property, $\gamma_1^i, \gamma_2^i, \dots, \gamma_n^i$ and also the reset

condition $\text{reset}(x)$ can only hold in t_i or t_j , but not in between. Hence, for the instances t_{i+1}, \dots, t_{j-1} , the signal x gets the value of the register x^{prv} assigned, the default case of the above definition. Thereby, x^{prv} has in each instant the value of x from the last instant:

$$x_i = x_{i+1}^{\text{prv}} = x_{i+1} = x_{i+2}^{\text{prv}} = x_{i+2} = \dots = x_{j-1} = x_j^{\text{prv}}$$

That means that in the instant t_j , the value of x^{prv} is the value that has been assigned to x in t_i . Consider now the delayed assignments and assume the case that no delayed assignment fired for the considered step, hence $\gamma_1^d, \gamma_2^d, \dots, \gamma_m^d$ are evaluated to false in the instants $t_i, t_{i+1}, \dots, t_{j-1}$. In this case, the value of x^{next} behaves as follows:

$$x_i = x_{i+1}^{\text{next}} = x_{i+2}^{\text{next}} = \dots = x_j^{\text{next}}.$$

The value of x from the instant t_i is assigned to x^{next} for t_{i+1} and is not changed for the remaining step. In the same way, x^{nas} is set to false and it is not changed in the step. However, this is exactly the behavior encoded by x^{nas} : if no delayed assignment is executed, it must be false. The crucial point here is that if no delayed assignment is executed in a step, the value of x^{next} and x^{prv} are the same, because both hold the value of x from the last step:

$$x_i = x_j^{\text{next}} = x_j^{\text{prv}}.$$

With this equivalence, the cases in the definition of the variable x can be joined in the same way as it was done for event variables:

$$\begin{array}{ccc} \vdots & \vdots & \vdots \quad \vdots \\ x^{\text{next}} : \text{clock}(x) \wedge x^{\text{nas}} & \implies & x^{\text{next}} : \text{clock}(x) \\ \text{trans}(x) : \text{clock}(x) & & \vdots \quad \vdots \\ \vdots & \vdots & \vdots \quad \vdots \end{array}$$

As in the case of the events, the register x^{nas} can be omitted and the resulting definition is equal to one given for events in E_2 (with the general definition of $\text{trans}_i(x)$).

In addition to the above Property P_1 , consider also a second one where the variable x is only read in instants of $\text{clock}(x)$. Hence, it is not read in instants of a lower clock and of course also not in instants of an unrelated clock:

$$\text{read}^i(x) \implies \text{clock}(x). \quad (P_2)$$

Hence, x is only *used* in instants in which $\text{clock}(x)$ holds. Note that the property can be also easily checked by the states occurring in the guarded actions and reset conditions that read the variable x . Looking at the definition of Equation E_2 above shows that if $\text{clock}(x)$ holds, the value of x is defined by the first cases, but never by the last one assigning x^{prv} . Since this value is not read, any value can be assigned in this case and the register x^{prv} can be completely omitted which results in the following equation:

$$\begin{array}{l}
\text{init}(x^{\text{next}}) = \text{default}(x) \\
\text{next}(x^{\text{next}}) = \begin{cases} \tau_1^d & : & \gamma_1^d \\ \tau_2^d & : & \gamma_2^d \\ \vdots & & \vdots \\ \tau_m^d & : & \gamma_m^d \\ \text{trani}(x) & : & \text{clock}(x) \\ x^{\text{next}} & : & \text{default} \end{cases}
\end{array}
\quad
\begin{array}{l}
x = \begin{cases} \tau_1^i & : & \gamma_1^i \\ \tau_2^i & : & \gamma_2^i \\ \vdots & & \vdots \\ \tau_m^i & : & \gamma_n^i \\ \text{default}(x) & : & \text{reset}(x) \\ x^{\text{next}} & : & \text{default} \end{cases}
\end{array}
\quad (\text{E}_3)$$

Register Definitions

Wire Definition

The last case is omitted and the second to last case is made the default one. Finally, the properties allow to reduce the complexity of the translation a lot. The first one allows to remove the Boolean flag counting the delayed assignments for memorized variables. The second property allows to also reduce the additional register used to store the value during substeps.

Note again that these properties are not artificial. They hold e. g. for variables of the lowest clocks, i. e. the leafs of the clock tree, and they also hold for variables which are not used together with refined clocks, but even if they are, if the communication happens in the *right* instants, the optimizations shown above can be applied. As an example reconsider the variables *a* and *b* in the GCD2 example in Figure 3.1 which are both only read in the first instant and then not used (read or written) in the substeps. Hence, the above optimizations can be applied to them.

Variable x is not written by Delayed Assignments

If the variable is not set by delayed assignments ($m = 0$), $\text{next}(x^{\text{nas}})$ is only assigned in Equation E₁ by itself and is initialized with *false*. Hence, x^{nas} can be substituted by *false*:

$$\begin{array}{l}
\text{init}(x^{\text{prv}}) = \text{default}(x) \\
\text{next}(x^{\text{prv}}) = x
\end{array}
\quad
x = \begin{cases} \tau_1^i & : & \gamma_1^i \\ \tau_2^i & : & \gamma_2^i \\ \vdots & & \vdots \\ \tau_m^i & : & \gamma_n^i \\ \text{default}(x) & : & \text{reset}(x) \\ \text{trans}(x) & : & \text{clock}(x) \\ x^{\text{prv}} & : & \text{default} \end{cases}
\quad (\text{E}_4)$$

Register Definitions

Wire Definition

Hence, if there are no delayed assignments, the two registers taking care of the delayed assignments can be omitted, since simply x^{nas} will never hold. More of interest is the case when there are no delayed assignments for an event variable in addition to the above shown Properties P₁ and P₂. In this case, the register occurring in Equation E₄ can also be omitted, since the value is not read in instances where $\text{clock}(x)$ does not hold, and the following equation is obtained:

$$x = \begin{cases} \tau_1^i & : & \gamma_1^i \\ \tau_2^i & : & \gamma_2^i \\ \vdots & & \vdots \\ \tau_m^i & : & \gamma_m^i \\ \text{default}(x) & : & \text{default} \end{cases} \quad (\text{E}_5)$$

Wire Definition

Hence, the event variable can be represented by a single wire whose value is computed in each instant based on the guarded actions which possibly set a value, or it is reset to its default value otherwise.

Summary

After having discussed some simplifications of the general hardware translation of the data flow, the properties and optimizations are now summarized. Therefore, the following table shows which equation can be used for a variable with the according property.

Properties	Event		Memorized	
	Equation	# Register	Equation	# Register
With Delayed Assignments	–	(E ₂)	(E ₁)	2 + 1
	(P ₁)	(E ₂)	(E ₂)	2
	(P ₁) and (P ₂)	(E ₃)	(E ₃)	1
Without Delayed Assignments	–	(E ₄)	(E ₄)	1
	(P ₁) and (P ₂)	(E ₅)	(E ₄)	1

The table also shows the number of the registers needed to represent a variable with a certain property in hardware. The most general case, Equation (E₁), requires 3 registers where two are of the same type as the variable itself and the third one is of Boolean type for x^{nas} . However, this third register is only needed for this equation. Of interest are the cases with the properties (P₁) and (P₂), because these properties hold for pure QUARTZ, hence for the single clock case, and the number of registers needed in this case is the same than for the original hardware synthesis. Furthermore, the properties also hold for variables of the lowest clocks (since there are no lower clocks which could read or write the variable), and for variables that are not simply used in the communication with variables of lower clocks. The overhead of the extension in terms of registers of the hardware translation is only present for variables that are really used in instances of lower clocks. If the communication does only take place in the *right* instances, there is no overhead (in terms of registers).

6.3 Scheduler

The scheduler triggers the execution of the actual system by setting the clock signals accordingly to a valid execution of the program. The constructivity that is considered in this

this is scheduling independent, hence each clock can be set accordingly to the control-flow locations. However, it is only allowed to set a clock, when a **pause** statement with the clock has been reached in the instant before, and no smaller clock can be also set. A clock that can be set according to the control-flow is called *enabled* and the condition for a clock C is defined as follows:

$$\text{enabled}(C) := \bigvee_{\ell \in \mathcal{L}, \text{clock}(\ell)=C} \ell$$

A clock is only allowed to tick when a label with the clock is active, hence at least one of the related **pause** statements was reached in the instant before. The relation to the definition of the semantics is that each clock contained in the set \mathcal{C} of the interpreter, is evaluated to true by this definition. Additionally, one of the smallest clocks must be chosen from this set, hence, the execution must synchronize at **pause** statements of a common clock. Therefore, the sets of all lower and all higher clocks of C are defined by:

$$\begin{aligned} \text{lower}(C) &:= \{c \in \mathcal{C} \mid c \prec C\} \\ \text{higher}(C) &:= \{c \in \mathcal{C} \mid c \succ C\} \end{aligned}$$

With these definitions, the equation for a clock can be defined as follows:

$$C = \underbrace{\text{enabled}(C) \wedge \bigwedge_{c \in \text{lower}(C)} \neg \text{enabled}(c)}_{\text{tick by its own}} \vee \underbrace{\bigvee_{c \in \text{higher}(C)} c}_{\text{tick forced by higher clock}}$$

A clock is set, when it either can tick by its own (it is one of the least enabled clocks), or when a higher clock ticks. This scheduler executes each instant as soon as possible (ASAP), and instances of unrelated clocks are also triggered in parallel. This is just one possible scheduler, but one that works for scheduling independent programs, hence, the scheduling restrictions considered come only from the control flow, but not from the data flow. Other schedulers could be thought of to target low power execution or resource sharing.

6.4 Summary

This chapter presented the translation of extended QUARTZ programs to synchronous hardware circuits represented by equations for wires and registers. The overall structure of the translation separates the functional part encoding the data flow and the control flow of the program from the scheduler triggering the actual execution by setting the clock signals. Thereby, the clock signals are logical signals obtained from the QUARTZ clocks and do not have any relation to the actual hardware clock of the synchronous circuit. The functional part was first developed as a general translation that can handle every variable in the intermediate format, which was then simplified for special cases based on the variables. The simplification is based on properties that hold e.g. for variables which are used like variables in pure QUARTZ and the complexity (in terms of used registers) was reduced to the same as for a hardware translation of pure QUARTZ. Hence, only variables used in complex interactions of clock bounds introduce more effort. The scheduler that was presented in this chapter is

rather simple, since the constructivity is scheduling independent, the scheduler can simply trigger the clocks as soon as possible. However, other execution schemes are imaginable, but the one presented turns out to be a valid one.

Chapter 7

Evaluation

After having explained extended QUARTZ with refined clocks, this chapter evaluates the achievements of the extension by addressing the applications and the relation to related work. Some example implementations using refined clocks are explained and compared to their pure QUARTZ counterparts. Afterwards, the related work already introduced in Section 2.1 is reconsidered and is now compared to the extension.

7.1 Examples

There are different motivations addressed by the extension. The first one is that computations that usually need more than one step can be hidden in a single step. The second one is that communication can now be implemented in a more flexible way and modules or threads only have to synchronize when data is exchanged. Furthermore, the extension also allows to hide the computation from the communication, and later changes of the timing does not change the behavior of the whole system. A JPEG decoder is introduced in the following to illustrate this motivations in a practical application.

7.1.1 JPEG Example

JPEG [Wall91, RoSa08a, SaMo10] was developed by the Joint Photographic Experts Group as a compression standard for digital image data and it is used in this work as a case study to illustrate the usage of refined clocks in the design of a decoding module. An overview of the compression method is given first, before the actual implementation is discussed. The JFIF file format [JFIF] is usually used to store the encoded image data, but it is only focused on the actual bit stream and the decoding here.

The JPEG standard defines several modes of operation for image encoding: *sequential*, *progressive*, *lossless* and *hierarchical*. The first two, sequential and progressive, are the practical relevant lossy compression methods based on the Discrete Cosine Transformation (DCT) [AhNR74, LoLM89]. Both differ in the way the encoded data is aligned in the binary data stream: for the sequential mode, the data belonging to the same image region is stored together, whereas for the progressive mode the data belonging to a certain image quality is stored together. Based on slow communication channels, it allows to decode a first (blurry) image from the received data and to enhance the quality when more data arrives. The

remaining two modes, lossless and hierarchical, did not gain much acceptance in practical applications. The lossless mode defines a different encoding allowing to reconstruct the *exact* image data from the encoded image, and the hierarchical mode allows to successively increase the resolution of the image, similar to the progressive mode. The lossy compression modes encode the actual image data by either Huffman encoding [Huff52] or by an arithmetic encoding, where also the second one is practically not relevant due to patents for this method.

Before the example implementations in pure QUARTZ and in extended QUARTZ for decoding an image based on the sequential mode are presented in the following, an overview of this encoding and decoding mode is given first. The decoding is easier to understand if the encoding is known, and therefore it is introduced first. The actual implementations are then explained without clock refinement, and then it is explained how the clocks can be used in the modeling of this example. Thereby, the explanation is as detailed as it is needed to understand the basic concepts of how clocks are used in this implementation, but not as detailed as it would be needed to understand the real encoding of each single bit.

Encoding

The sequential mode of image encoding as defined in the JPEG standard is explained in this section. The actual image data is usually given as an RGB image where each pixel is represented by three values for its red, green and blue channel. Since encoding is based on the $YCbCr$ color space representing a pixel by three values for the luminance (brightness), the red difference and the green difference, the image must be converted to this color space. The three channels of $YCbCr$ are then encoded separately.

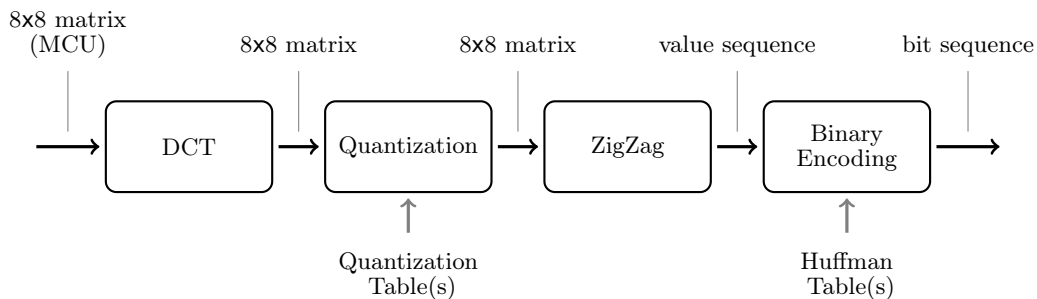


Fig. 7.1. JPEG Encoding Chain

Each channel of the image is divided into a list of 8x8-blocks called Minimum Coded Units (MCUs) being encoded separately. An MCU does not necessarily represent a block of 8x8 pixels, because subsampling can introduce a scaling of the actual pixels to these blocks. However, this case is not considered here, but it could be achieved by an additional clock at the beginning of the presented encoding chain. There is also a virtual extension of the actual image defined, if its boundaries do not exactly fit into the grid of 8x8-blocks. However, the presented encoding starts with the sequence of MCUs which have to be extracted from the actual image before in either way.

The encoding chain for the MCUs is shown in Figure 7.1. It starts, as explained, with a sequence of 8×8 matrices representing MCUs. A matrix is transformed by the (two-dimensional) DCT to the frequency domain resulting in another 8×8 matrix. The matrix does no longer contain pixel values, but coefficients of frequencies that can be used to rebuilt the pixels by using the Inverse Discrete Cosine Transformation (IDCT). The first value of a matrix in the upper left corner is called the DC (*direct current*) value representing an average of all pixels. The other 63 values are the AC (*alternating current*) values with the *real* frequencies. The more the value is in the lower right, the higher the frequency for this coefficient is. The two-dimensional DCT for a matrix M can be mathematically described as two matrix multiplications $C \times M \times C^T$ with a matrix C containing constants.

The second step of the encoding is the *quantization* where the coefficients obtained from the DCT are scaled down by a certain factor. The result is another 8×8 matrix of the same size, but with smaller values. At the first view, it seems that there is no information lost during these first two steps, since both can be exactly inverted. But this is only the case if they are performed on real numbers. So, dependent on the implementation, the DCT is performed on integers or on floating point numbers, but right after the quantization the values are converted to integers to be encoded in the following. So, scaling down loses information, but having small values, or at least often the same values, leads to a good compression of the data. Also, a lot of coefficients obtained from DCT are near to 0, thus scaling down sets them to 0 which is a small change that is usually not *visible* in the resulting image.

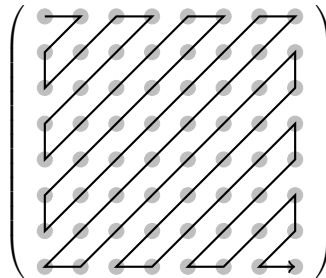


Fig. 7.2. ZigZag Ordering

After having the 8×8 matrix scaled down, it is converted to a stream of values. The sequence of the values in the matrix is given by the ZigZag module ordering the entries as shown in Figure 7.2. The ordering is also useful for a good compression since it aligns coefficients of *similar* frequencies next together which will produce sequences of 0s which are encoded very efficiently.

In the last step, the sequence of 64 values is encoded into a bit sequence. The values are grouped in the sequence accordingly to preceding 0, and the actual non-zero values are length-dependently encoded. Such a group consists of three values $((l, s), b)$, where l is the number of leading 0s and s is the number of bits needed to encode the actual value represented by the bit sequence b . For example, consider the following sequence:

$$\underbrace{6}_{((3),110)}, \underbrace{0,0,0,0,3}_{((4,2),11)}, \underbrace{0,-5}_{((1,3),010)}, \dots \longrightarrow ((3), 110), ((4, 2), 11), ((1, 3), 010), \dots$$

The DC value is handled differently, because it is the first value in the sequence and has no leading 0s. The value 3 in the sequence has 4 leading zeros and the value itself is encoded according to the JPEG standard to the bit sequence 11. For the encoding, the pair (l, s) is encoded by the Huffman encoding and the bit sequence is used directly.

The quantization tables and the Huffman tables for encoding and decoding are contained in the image metadata and stored in the JFIF file. Some tables are predefined in the JPEG standard, but any other table can be used and they are usually given in the file. The quantization table is responsible for the compression rate and also for the loss of information.

Decoding

Technically, the image decoding works the other way around, but the understanding is easier when it is known how the bit sequence is produced. The decoding chain is illustrated in Figure 7.3, but for the implementation the order of dequantization and DeZigZag module was changed, since each decoded value can be scaled by its own before it is arranged in the matrix.

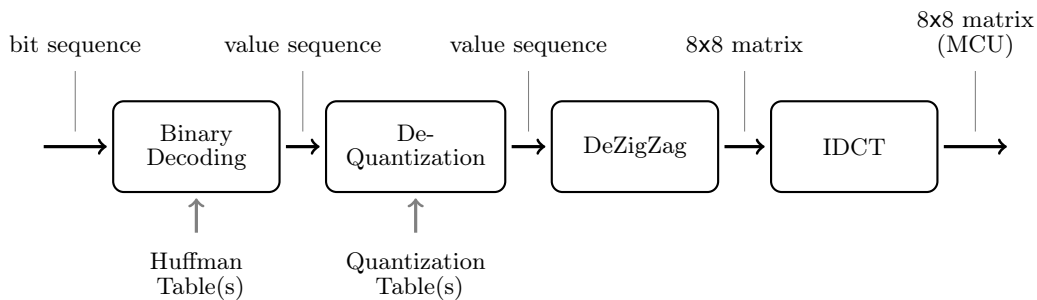


Fig. 7.3. JPEG Decoding Chain

The binary decoder reads the binary image data by first decoding a pair containing the number of leading 0s and the number of bits for the actual value with the given Huffman table. Based on these values, the actual data is decoded from the stream and the sequence of 0s followed by the actual value is produced. Afterwards, the next pair is decoded by the Huffman decoder. The DeZigZag module reorders the value sequence to an 8x8 matrix where each value has already been scaled-up by the dequantization module. Afterwards, the IDCT converts the coefficients back to the MCU. The IDCT is defined for a matrix M by the multiplication $C^T \times M \times C$ with the same matrix C used for the DCT. Finally, the resulting MCU is obtained.

Implementation in Pure QUARTZ

This section explains the implementation of the JPEG decoder in QUARTZ covering the above introduced decoding chain. It decodes the actual binary image data into MCUs and requires the Huffman and dequantization tables as parameters. The tables can be extracted from a JFIF file, or they can be fixed for a certain application, e.g. for decoding an image stream from a camera which uses the same tables for encoding every time.

Recall the fact that a strictly synchronous system as it is described by a QUARTZ program gets one value per input and produces one value per output in each step and that there is no notion of presence or absence. Hence, data-flow applications, with different input and output rates cannot be directly described. As an example, consider the binary decoding which is the first module in the chain. It decodes one byte from the input bit-stream containing the number of leading zeros and a number of bits for decoding the actual value. Depending on those values, from 1 to 16 values (leading zeros + actual value) are produced and given to the next stage. Since the Huffman encoding is of variable length, the binary decoding module has no input/output relation in the form $1 : n$ or $m : 1$, but it has the relation $m : n$. The other modules are simpler, the dequantization reads one value and produces one, and the DeZigZag reads 64 values in sequence and produces one 8×8 matrix. Finally, the IDCT reads one matrix and produces one.

Since QUARTZ does currently not have any constructs to model the data-flow-style communication between the modules, the communication is made explicit. Therefore, the availability of new data for the successive module is notified by additional *ready* signals. In the following, the implementation of each module is explained.

Binary Decoder

Technically, the binary decoder consists of two modules, the Huffman decoder and the value decoder. The first one decodes one byte from the binary input stream based on the given Huffman table. The byte is then interpreted as a pair of values by the value decoder. Based on the first entry a (possible empty) sequence of leading zeros is produced which is followed by the (real) value being read from the input stream based on the second entry of the pair. The composition of the binary decoder and its data flow is illustrated in Figure 7.4 where also example values are shown. Assume that the first bits 0, 1, 0 from the input are decoded by the Huffman decoder to the pair (3, 2) represented in one byte. The actual result of the decoding here depends on the given Huffman table. Based on the first entry of the pair, the value decoder produces 3 leading 0s for the output. Afterwards, it reads 2 bits from the input stream based on the second entry of the pair being then decoded to the value -2 . The decoding is defined in the JPEG standard and does not depend on any other information. For the first value of each block, the DC value, no leading zeros are processed.

Both modules, the Huffman decoder and the value decoder, need to read bitwise from the input stream, since especially for the Huffman decoder the length of the encoded data depends on the data itself. In the implementation, a QUARTZ module `readBit` provides access to the next bit of the stream and handles the communication. Its interface is given by the code in Figure 7.5. The module `readBit` reads byte-wise from the input stream and provides the request for the next byte by the signal `byte_request`. The byte itself is cached until all bits have been read. However, reading a bit requires one step since the internal state

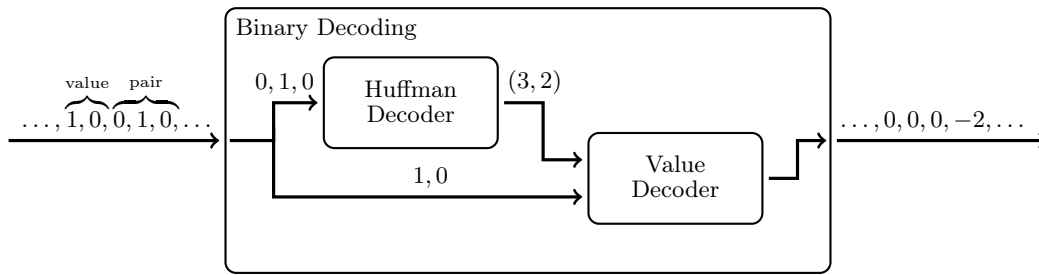


Fig. 7.4. Binary Decoding

of the module must be updated and potentially a new byte must be requested. The module `binDecode` has the interface that is shown by the code in Figure 7.6. Hence, the tables are passed through to the Huffman decoder and the interface needed for `readBit` is also. When a value was decoded, it is provided through the variable `value` and `value_ready` indicates that a new value is available.

```

module readBit (
  // input interface
  event ! byte_request,
  bv{8} ? byte,
  // cache
  nat{8} bit_index,
  // provide data bit
  bool ! bit
) {
  ...
}

```

Fig. 7.5. Module `readBit`

Dequantization and DeZigZag

The next two stages are quite similar and implemented by the QUARTZ modules `deQuant` and `deZigZag`. Both wait for the occurrence of an input value and when it arrives it is processed. The `deQuant` module can directly provide the scaled output value, whereas the module `deZigZag` waits for 64 values until the whole block is provided. The structure is illustrated by the code stub in Figure 7.7 for the module `deZigZag`.

Inverse Discrete Cosine Transformation (IDCT)

The last module in the considered decoding chain is the module `idct2d` transforming a given matrix. A simple implementation for the IDCT algorithm is by two matrix multiplications, but due to the structure of the matrix a lot of optimizations have been developed in the past [FeWi92, DiWa04]. The version used here is based on a 1-dimensional IDCT similar to

```

module binDecode (
  // the huffman tables for dc and ac values
  ([16]nat{256} * [256]bv{8}) ? huffman_table_dc,
  ([16]nat{256} * [256]bv{8}) ? huffman_table_ac,
  // interface to read binary data from
  event ! byte_request,
  bv{8} ? byte,          // current byte
  nat{8} bit_index,
  // output the decoded value
  int{32768} ! value,
  event value_ready
) {
  ...
}

```

Fig. 7.6. Module binDecode

```

module deZigZag (
  // input interface
  int{32768} ? in_value,
  event      ? in_value_ready,
  // output interface
  [8][8]int{32768} ! out_block,
  event
! out_block_ready
) {
  ...
  loop {
    await(in_value_ready);
    ...
    if(index == 63) {
      emit next(out_block_ready);
      ...
    }
  }
}

```

Fig. 7.7. Module deZigZag

the one published in [LoLM89] and requiring 11 multiplications that is applied to the rows and the columns of the matrix. Hence, $(8 + 8) * 11 = 176$ multiplications are needed. The basic structure of the module is illustrated by the code in Figure 7.8. The module `idct2d` applies the 1-dimensional IDCT to each row and each column, where one step is needed per row and per column.

```

module idct2d([8][8]int{S} ?mi, [8][8]int{256} mo)
{
  ...
  // transform rows
  while (row < 8) {
    next(row) = row + 1;
    idct1dRow(mi, mo_tmp, sat{8}(row));
  }
  // transform columns
  while (col < 8) {
    next(col) = col + 1;
    idct1dCol(mo_tmp, mo, sat{8}(col));
  }
}

```

Fig. 7.8. Module `idct2d`

Everything Together

Finally, the modules are put together in parallel to decode a sequence of MCUs for the whole JPEG image. The code for this is shown in Figure 7.9.

```

binDecode(...);
||
deQuant(...);
||
deZigZag(...);
||
loop {
  await(block_ready);
  idct2d(...);
  emit(mcu_ready);
}

```

Fig. 7.9. Code of JPEG Decoding Chain

Implementation with Refined Clocks

After having explained a possible implementation in pure QUARTZ of the JPEG decoding chain, this section introduces the structure of an implementation in extended QUARTZ. Recall that, the extension was introduced to (1) hide internal computations and (2) avoid unnecessary synchronization, but it still keeps the synchronous model at each abstraction level. With these premises, it is explained in the following how this concepts can be used in the modeling of the JPEG decoding chain. Thereby, clocks are used to synchronize modules when data is exchanged, but also to do computation steps independently. Please note that the design flow of first writing a pure QUARTZ program and then translating it to the extension is not the usual development flow that should be taken. It is only used here for comparison, while the usual process should rather use the features of the extension from the beginning.

The module `deZigZag` collects 64 values which are then organized in a matrix to the `idct2d` module. Hence, synchronization between this module is only needed when a whole block is completed. The other modules only communicate single values, hence, they only need to synchronize when a value is available. Finally, the modules `binaryDecoder` and `idct2d` do computations which are not finished in one step, hence, additional clocks are used to hide their computations.

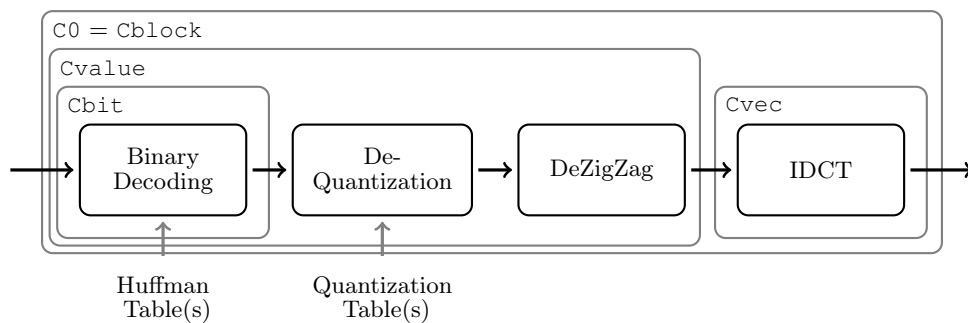


Fig. 7.10. Clock Boundaries for JPEG Decoder

The structure of introduced clocks and the modules are shown in Figure 7.10. The clock `Cblock` is used for the communication of 8×8 blocks and synchronizes the modules whenever a block is exchanged. It is also the outermost clock for this example, and therefore it is set to the module clock `C0`. The module `idct2d` computes the IDCT when a new block arrived and it does not need to synchronize until it finished its computation. The clock `Cvec` is used inside the module for the independent computation and it processes a vector (row or column) in each step of the clock. Dependent on the used algorithm for IDCT or another granularity of the description, other partitions are possible. The communication of single values between the modules `binDecode`, `deQuant`, and `deZigZag` is synchronized by the clock `Cvalue`, hence in each step of the clock, a value is communicated. Finally, the clock `Cbit` in the module `binDecode` is used to read bitwise from the binary input data and to decode the data values.

However, the extension introduces several abstraction layers into the synchronous model, but keeps this model at each level. Therefore, communication with the environment is only

possible with variables of the module clock, which means that all data that is processed during substeps must be available for the whole step. The module `readBit` that is used inside of the binary reader cannot request a new byte from the environment each time it needs one, since the hierarchical structure of the clocks does not allow this communication. Instead, the whole data that is possibly needed to decode a whole block must be provided to the module. Consider first an upper bound of encoded data for a block (of images with 8 bit precision):

$$\underbrace{64 * 16 \text{ bit}}_{\text{Huffman encoded}} + \underbrace{1 * 13 \text{ bit}}_{\text{DC data value}} + \underbrace{63 * 12 \text{ bit}}_{\text{AC data values}} = 1793 \text{ bit} \approx 224 \text{ byte} < 256 \text{ byte}$$

A block consists of 64 values which are in the worst case not 0, and each value is Huffman encoded with an improper Huffman table by at most 16 bits. Based on the Huffman decoded value for each one of the AC values, 12 additional bits and for the DC value 13 additional bits are read. Hence, an upper bound is given by 256 bytes which can be considered as a reasonable cache size. Note that the final MCU only contains data of 64 bytes, hence, this worst case encoding will probably not occur in any real applications. The interface of the module `readBit` is changed as it is shown in Figure 7.11 and it requests new bytes instead of just a single one. The received bytes are cached and the index of the byte and of the bit which should be read next is stored. Since the module is called at two places in the binary reader, this state must be handled by the binary reader itself and is provided for each call. Finally, the whole computation can be put together by the code given in Figure 7.12. The local clocks `Cbit` and `Cvec` are defined in the modules and are not shown in this complete structure, whereas the clock `Cvalue` is defined here for the communication of single values. The clock `C0` used for synchronizing the communication of blocks is the module clock and is defined implicitly. Finally, the *ready* signals that notified the availability of new data are no longer needed, since the clocks are utilized for this synchronization. After a translation to hardware (or software), these clocks behave very similar to the ready signals, but the difference is that the communication is now made explicit and when no communication happens, the modules can run independently.

Comparison

The JPEG example also shows a disadvantage of the extension so far: the data that is required for processing in substeps must be completely available at the beginning. Hence, for decoding a whole block in one step at a certain clock level, the decoded data must be completely given, and for computing the IDCT of a matrix, the whole matrix must be available. When the abstraction level is raised to e. g. decode the whole image within a step, the whole decoded image data must be given at once. Furthermore, the example also shows that new clocks are rarely used for the implementation.

Two more remarks should be given on the actual implementation. First, for the pure QUARTZ version, an additional array between the module `DeZigZag` and `IDCT` is needed to buffer a decoded block. When a block is produced, the `IDCT` starts, but the decoding also proceeds with the next block and overwrites the values in the array. To ensure that the values are not changed until the `IDCT` module reads them, it must be copied. This error did not occur in the first implementation, but after changing the `IDCT` to use more steps to shorten

```

module readBit<Cbit> (
  // input interface
  event ! byte_request,
  [256]bv{8} ? bytes,
  // cache
  [256]bv{8} cache,
  nat{256} byte_index,
  nat{8} bit_index,
  // read bit
  bool ! bit
) {
  ...
}

```

Fig. 7.11. Module readBit with Refined Clocks

```

clock(Cvalue) {
  ...
  binDecode{Cvalue}(..., byte_request, bytes, value);
  ||
  deQuant{Cvalue}(dq_table, value, qvalue);
  ||
  deZigZag{Cvalue}(qvalue, out_block);
}
||
loop {
  pause;
  idct2d(block, mcu);
}

```

Fig. 7.12. Code of JPEG Decoding Chain with Refined Clocks

the critical path in the hardware implementation as it will be explained in the next section, the results became wrong because the DeZigZag module overwrites the values too fast. The additional array ensures this, but the interesting thing is that this array is automatically created by the synthesis for the extended QUARTZ version because delayed assignments are here buffered until the next step of the clock starts, which is the next synchronization when the next block is exchanged in the example. A second remark refers to the constructivity definitions given in Section 4.5, where the restriction is that a variable is only immediately written in an instant of its clock. The JPEG example naturally follows this approach, even without spending too much effort to achieve this. Furthermore, also the restrictions defined for the loops that can be compiled without duplicating the loop body (Section 5.3.3) are no problem: the example also obeys these restrictions. Hence, the restrictions and definitions turn out to be usable for programming, since for real-world programs a very structural way

of description is usually used. The rare use of new clock definitions (4, respectively 6 as they are used for the experiments in the next section) emphasizes this.

7.1.2 Experimental Results

The experimental results obtained from some programs including the JPEG examples are presented in the table in Figure 7.13. First of all, note that the extension was introduced to provide more flexibility in the source language for writing programs. Furthermore, pure QUARTZ programs and extended QUARTZ programs are compared based on the translation to hardware which relies on the equation system shown in Chapter 6. Since such an equation system represents a synchronous hardware circuit and as such it can be also directly translated to pure QUARTZ, the obtained hardware could be also written in QUARTZ. However, independence and synchronization must be *explicitly* programmed and cannot be left for the compiler like for the extension. Hence, the obtained hardware from an extended QUARTZ program is not expected to be faster or smaller, but it is more flexible in the source language and also in execution (e. g. by using different schedulers). Furthermore, the aim of the examples is to show the difference between pure QUARTZ and the extension, and not to provide an optimal implementation of an algorithm.

Example		QUARTZ		Equations		Circuit		
		# Clocks	# LoC	# Reg.	# Wire	# Reg.	# LUTs	Delay
JPEG	single	1	$\sim 1K$	307 + 31	180	9,132	9,049	20.2ns
	ext.	6		374 + 29	191	11,208	11,812	26.0ns
IDCT2	single (1)	1	~ 350	130 + 4	184	3,995	4,047	25.7ns
	single (2)	1		148 + 18	144	4,973	6,780	11.4ns
	ext. (1)	2		130 + 4	188	4,018	4,052	25.9ns
	ext. (2)	4		150 + 18	152	4,606	6,233	12.9ns
GCD	single	1	15	3 + 2	3	33	104	3.9ns
	ext.	2	17	3 + 2	7	33	78	3.7ns
TRACE	single	1	~ 100	10 + 11	26	57	121	5.8ns
	ext.	3		17 + 14	32	76	180	6.5ns

Fig. 7.13. Experimental Results

The table compares implementations in pure QUARTZ with implementations in extended QUARTZ. Thereby, basically three areas are considered. The first one shows the statistics of the source code in (pure and extended) QUARTZ to get an impression of the size of the example. The second one shows the statistics of the equations based on the translation presented in Chapter 6. The number of registers consists of the number of arbitrary *scalar* (an array consists of multiple scalar registers) registers and the number of Boolean registers (the second number). Furthermore, the number of wire definitions is shown. Based on the

equation systems, VERILOG code is generated which is then synthesized to an FPGA leading to the statistics for the final circuit: the number of the required 1-bit registers, the number of the Lookup-Tables (LUT), and the minimum clock delay are shown. For the translation to VERILOG, some adaption is needed. First, VERILOG does not support multi-dimensional arrays and therefore the 2-dimensional arrays are first translated to 1-dimensional arrays. Second, the translation of the type system causes some overhead due to the adjustment and representation of the operators and types. However, since the extension is compared with pure QUARTZ, the same overhead is produced here for both.

The synthesis tool was used to optimize the speed of the resulting circuit. The program GCD is the example that was used to introduce the extension in Section 3.2. The extended version needs two wires more than the pure QUARTZ version due to the clock definitions. However, the resulting hardware does not produce more overhead. Instead, the resulting clock delays are pretty close and the examples require less than 1% of the FPGA. Hence, this result can be justified by the unpredictability of the synthesis algorithm.

The TRACE example is of interest, because it tests variables which are read and written at arbitrary clocks levels. Hence, it tests the translation of all cases explained in Section 6.2. The single-clock version of the module is basically not really useful, because it cannot represent the behavior, but it is provided here to show the overhead that comes from the usage of variables at different clock levels. Besides the data variables, the example contains 11 labels resulting in Boolean registers. Without taking the labels into account, the overhead is around 100% for the equations. However, the example is used as a test case and uses all arbitrary accesses of variables. The overhead in terms of the circuit is not as big, but optimizations due to the actual used bits can reduce the number of registers here. The clock delay leads to the expected result that due to the increased overhead also more time for the computation is required.

The IDCT2 module is compared based on two versions. The first version computes the 1-dimensional IDCT in one step for a row or column of the matrix. This leads to a huge critical path containing several multiplications and requiring a long clock delay. The second version adds more substeps of new defined clocks, one for the rows and one for the columns, for computing the result. In this way, the multiplications are separated leading to a shorter critical path. The same change is also done for the pure QUARTZ version, but there are real macro steps added to separate the multiplications: the behavior of the module is changed. A multiplication requires with the used bitwidth around 3ns to 4ns, which is still far away from the clock delay, but as already said, some overhead is caused by the translation of the type system. The examples show that the pure QUARTZ version is slightly faster than the extended version, and that they are close in terms of required resources. However, the synthesis algorithm produces some more overhead for the second version for pure QUARTZ.

The JPEG example requires a total amount of 6 clocks in the extended QUARTZ version because the second version of the IDCT2 is used. The difference in the registers and the wires for the equations comes basically from the module readBit which needs to buffer bytes in contrast to the pure QUARTZ implementation where a new byte can be requested every time. The critical path of both implementations is the same data path, but for the extension it is more complex due to the more control signals (e. g. clock wires). The example shows that an overhead of around 20% to 30% is a realistic assumption for the resulting hardware.

Finally, the examples show that there is some overhead introduced by the extension in terms of the hardware translation. The additional control wires for clocks and labels seem to not significantly influence the hardware size, but some more logic is needed. When the overhead is explicitly provoked like in the `TRACE` example, it is there, but in the other examples, it seems that it is amortized by the other logic. However, the effects could be different, if the type system would be translated in a more efficient way.

7.2 Comparison with Related Work

The clock refinement extension to `QUARTZ` was introduced to provide more flexibility (especially in *timing*) for writing programs in an imperative synchronous language. Additionally, the extension was designed to *keep* the synchronous abstraction at each layer: each step of a clock behaves like a macro step of a module on this level. The hierarchy of clocks does only allow structural refinement and synchronization. On the one hand, this extension is the first one of this kind to synchronous languages, but on the other hand, there are other concepts and models of parallel compilation that behave similarly. These are discussed in this section with respect to the extension.

7.2.1 ESTEREL

The first language that is compared to the extension is `ESTEREL` (Section 2.1.1) because it is also an imperative synchronous language, and it has a feature which `QUARTZ` does not have. The variables in `ESTEREL` can be assigned multiple times in a macro step, and reading them results in the last written value. To guarantee determinism, it must be ensured that read and write accesses to each variable are totally ordered. This is ensured in `ESTEREL` [Berr00] by forbidding reading and writing of variables in different threads: if one thread writes a variable, this variable is only allowed to be used in this thread. However, a variable can be set before, and it can be used in read-only mode in parallel threads. In this way, read and write operations are limited to a sequential execution which also orders the accesses (note that multiple read accesses as they can occur in parallel threads do not have to be ordered).

There are similarities between `ESTEREL` variables and variables of refined clocks in `QUARTZ`: both can have multiple values during a macro step, respectively in a step of the higher clock. However, since loops with an instantaneous loop body are (for good reasons) forbidden in `ESTEREL` (and also in `QUARTZ`), there can be only a finite number of values of a variable in `ESTEREL`, and the number is determined by an upper bound based on the assignments occurring in the source code. To be fair, there is also a well-defined upper bound for values of a variable in the extension, since it must be ensured that each macro step of the module clock terminates (to ensure reactivity), and since data types of infinite size are not considered for designing real systems. However, the `ESTEREL` variables provide one abstraction level, but due to the clock declarations, the extension can provide arbitrarily many of them. Thereby, the variables can be also used in parallel threads where the `pause` statements related to lower clocks ensure the total order of read and write accesses.

7.2.2 Multiclock ESTEREL

The multiclock extension to ESTEREL (Section 2.1.1) has been introduced to model ESTEREL processes that do not run synchronously, i. e. they do not synchronize each macro step, but each one is triggered by its own clock. The interesting part of this extension is that even if processes run on different clocks, they can be nested by module calls. Hence, a module can call another one running on a different clock, and communication is established by well-defined communication devices basically transferring the values across the clock boundaries. A structural hardware translation of ESTEREL [BeHH92, Berr92] where each statement is separately translated and connected by control-flow signals similar to the predicates used in the QUARTZ compiler (Section 2.2.6). These signals also *start* submodules and submodules tell when they are *terminated*. In this way, the multiclock extension can be seen as a separate translation of the individual modules, where also the control signals are transmitted by the communication devices across clock boundaries. This results in individual modules running on their own clocks like it was illustrated in Section 2.1.1. The clocks triggering the module execution are thereby external signals which can be periodic hardware clocks.

Hence, the modules can run at their own speed which can be either determined by clocks triggering the executions for example in hardware, or by just running as fast as possible when they are for example translated to software. In fact, the modules can wait until a value that is written by another module changes, but there is no means to implement any synchronization by dedicated constructs. In contrast, the clocks of the presented clock-refinement extension are not coming from the outside of the system to trigger the computation at arbitrary speed. They are used to refine already existing computation steps to determine synchronization points and when values of variables change. Thereby, they keep the synchronous model at each abstraction level and also called modules behave like synchronous models according to their own clock.

7.2.3 SIGNAL

The polychronous language SIGNAL (Section 2.1.1) allows to specify the behavior of systems in a data-flow manner. Since the language is used for *specifying* the behavior, the compiler has to translate the system to (operational) executable code.

An interesting feature of the language SIGNAL is *oversampling* which allows to run one computation *faster* than the other one. For this purpose consider the GCD example for SIGNAL, which is given in Figure 7.14, and which is very similar to the Counter that has been shown in Section 2.1.1. In the first instant, the process gets the values for *a* and *b*, and the operator `init` transfers them to *x* and *y* for the next instant of their clock. The signals *x* and *y* are defined so that they compute the GCD by the usual Euclidean scheme: due to the `init` operator, the next value of *x* is based on the condition $x \geq y$ either defined by $x - y$, or the value of *x* is kept. Finally, when the GCD is computed, i. e. either *x* or *y* is less or equal to 0, then the computed value is written to the signal `gcd`. The additional clock constraint ensures that input values for *a* and *b* are only accepted when the computation is finished (or the whole process starts). An outer process using this computation can only synchronize with the GCD module when inputs are given, and when the output is computed. Thereby, the outer module does generally not know how much instants of the clock of *x* and *y* are needed for computing the GCD, but the synchronization ensures that it will not miss the

```

process GCD =
(? integer a, b;
 ! integer gcd;)
(| x := (a default ((x-y) when (x>=y))
           default x) $ init 0
 | y := (b default ((y-x) when (x<y))
           default y) $ init 0
 | gcd := y when (x<=0) default
           x when (y<=0)
 | a ^= b ^= (when (x<=0)) ^+ (when (y<=0))
 |)
where
  integer x, y;
end;

```

	1	2	3	4	5	6	7
a	7						4
b	3						2
x	0	7	4	1	1	1	0
y	0	3	3	3	2	1	1
gcd	0						1

(a) Code

(b) Trace

Fig. 7.14. SIGNAL Example: GCD

result. The trace shows a sample execution of the process, where the values 7 and 3 are given as inputs, and the computation is processed while the values of *a* and *b* are absent. When the computation is finished, the output is provided and new inputs can be given.

The example looks very similar to the introductory example used to explain the clock refinement extension in Section 3.2. After the input values are provided, the computation can run independently and then provides the result. Thereby, the model specifies *when* the result is given. For the SIGNAL example, this is the *next* instant when communication with the process takes place. For the extension, the GCD computation is finished in one step independent of the actual number of substeps it takes. This example shows also the difference here, where the extension can provide the result in the same step, SIGNAL provides the result in the next synchronized instant. In Section 4.5, a special abstraction for checking the constructivity of programs with refined clocks were introduced, which restricts the program that variables of clock *c* can be only written by immediate assignments in an instant of clock *c*, but they can be written by delayed assignments in between. With this restriction, the GCD example for the extension cannot provide the result in the same step, but only in the following one. Hence, the notion of constructivity coincides with the SIGNAL example, where the result is given in the next synchronized instant.

However, even if SIGNAL is less expressive in this sense (oversampling result is available in next instant), it also has advantages over the extension. The synchronization in terms of common clocks is not structural like in the extension and therefore, each variable can be synchronized with each other. Therefore, it is possible that three processes communicate (each one with each other one) based on three independent clocks. A situation that is not possible in the extension. Remember also the `Counter` example from Section 2.1.1, which produces for each given input a sequence of outputs. Hence, the outputs can be produced *faster* than the inputs (also the other way around is possible). Obviously, also the $m : n$ relation of inputs and outputs can be achieved. This freedom, as it was also explained for the

JPEG in Section 7.1.1 is not possible for QUARTZ and also not for the extension at module boundaries.

Furthermore, in SIGNAL, the clock of a signal identifies the instants in which it has a value, and in all other instants, the signal is absent: it cannot be read or written. Thereby, the clock of a signal can be defined by its definition ($x := \dots$), by its use in expressions ($\dots + x$), or by additional clock constraints ($x \hat{=} \dots$). Reading a signal in an instant forces it to be present, and then a value must be given. By additional macros, which can be rewritten with the SIGNAL primitive statements, a variable can be also read when it is not present: `var x init c`. The macro creates an explicit register to *memorize* the last value of x , and initializes it with c . However, this is similar to the additional registers needed for the hardware translation of the extension when a variable is read in instants of a lower clock (cf. Section 6.2.4), but with the difference that the register is automatically created for the extension if it is needed. Compare it again with the given restriction for the extension that a variable of clock c is only written in instants of clock of c . In this case, both languages are again similar: The value of a variable can only change when its clock holds, but it can be read in all other instants (either by `var x init c` in SIGNAL, or with just x in the extension) leading to the value that has been finally assigned.

Besides some similarities, the languages also have some differences: in the extension the clocks are determined by the control flow of the program and not by the data flow of the variables. Due to this fact, it would be not wise to forbid reading a variable when its clock does not hold, and the synthesis cares about storing the value if it is needed. In SIGNAL, a clock notifies about the presence, whereas in the extension it generally identifies two instants between which the value of the variable does not change (resp. is set once). The differences here came from the different approaches: control-flow and data-flow driven execution.

7.2.4 LUSTRE

The synchronous data-flow language LUSTRE looks similar to SIGNAL, but there are some differences. The main difference is that in LUSTRE, the clock of a signal is completely determined by its definition, and not by its use. Hence, there is no possibility to add additional clock constraints, and the clock is also not determined by the usage of a variable in expressions ($\dots + x$). Instead, in such functional expressions, only signals of the same clock can be combined. The clock of a signal can be derived from the values of existing signals with the `when` operator. Since the clock is only derived by already existing clocks, the clocks can be arranged in a tree where the root clock must be in the interface of a LUSTRE node: each computation is triggered by inputs. Thereby, oversampling where local clocks run *faster*, is not possible. However, arbitrary clocks at the interface are possible, as long as they can be determined by other input signals.

7.3 Summary

This chapter evaluated the achievements of the extension by considering the modeling of a JPEG. Furthermore, results from the hardware synthesis were presented to see the overhead of the extension. Finally, the related work was compared to the extension.

Chapter 8

Conclusion

This thesis introduced clock refinement as an extension to imperative synchronous languages. It came from the simple idea to divide existing macro steps into smaller substeps which by themselves behave like macro steps. The Euclidean algorithm to compute the GCD of numbers was used to illustrate the extension based on the imperative synchronous language QUARTZ. Even though the idea is rather simple, it has some impacts on the whole language because the synchronous assumption is kept also for the substeps, and therefore, the changed behavior of the statements of QUARTZ were discussed. The synchronous assumption comes along with the discussion of causality and constructivity. Since all actions are logically executed in a single point of time, the dependencies between them have to be considered to get a practical notion of executing programs (in contrast to logical correct where all solutions are allowed). This discussion was extended to the extension and an informal view on the execution of substeps was developed.

The semantics of the extension was defined in the same way as for pure QUARTZ with two sets of SOS rules: the transition rules and the reaction rules. In contrast to the original rules, these rules are defined to handle various clocks, and especially the transition rules also have to be defined for partial environments since values might be not known at the beginning of a step, or they belong to variables of an unrelated clock. Furthermore, variables can only change for steps of their clocks, and also abortion and suspension are bound to the clock level the statements are defined on. The interpreter that combines the rules to an executing machine deals with two additional issues: first, it must select a clock to execute the next instant, and second, it must decide when the default reaction for a variable can be executed. The first issue arises with unrelated clocks which are defined in threads of the parallel statement, and the threads can execute unrelated steps. However, each choice that leads to valid execution does also lead to the unique behavior of the program with respect to the environment. Furthermore, it turned out that the choice of the clock cannot lead to an invalid execution if no immediate data dependency exist between the unrelated clocks (scheduling independence). The second issue arises with immediate assignments of variables in a substep of a lower clock. Since the variables have one unique value for a whole step of their own clock, it cannot necessarily be decided what the value is known at the beginning of the step. Two notions of constructive executions have been defined. The first one does not allow to immediately assign variables in substeps of a lower clock, and the second one just requires the natural order: the variable must be assigned before it is used. Even though, the second

one is given in a declarative way for the interpreter, under the assumption that the property is given, it can be implemented efficiently as it was shown in the hardware translation.

The compilation algorithm translates programs to an intermediate format. Thereby, it also has to deal with schizophrenia problems arising with local declaration in imperative synchronous languages. An intermediate format was given and it was discussed that the original solutions used for the translation of pure QUARTZ to an intermediate format are not applicable for the extension. Therefore, a solution based on a special case was presented that allows an efficient translation. Other cases can still be handled by the approach to duplicate the body of the loop in the source code. Furthermore, like one can abstract the notion of constructivity in pure QUARTZ (or other synchronous languages) by checking for acyclic dependencies between the variables, the same can be done in the extension based on the intermediate format: the control flow has to be also considered to decide which actions can be possibly reached in a steps, but as long as no write action for a variable can be reached after a read action in the control-flow graph, the program is constructive and can be implemented.

Furthermore, a hardware translation based on the intermediate format was presented. Thereby, each variable is translated separately to equations which can then be translated to a hardware description language. The equations were first presented for an arbitrary case and then some special cases were introduced. These cases occur e.g. when a variable is not immediately written in substeps. The properties allow to reduce the complexity of the resulting equation system, and they apply to variables which are not used in the data flow of substeps. Furthermore, the properties also hold for pure QUARTZ and it turned out that in this case the translation to hardware is mostly the same. Hence, there is some overhead produced in the hardware translation, but it is only existent when a variable is used in substeps. In addition to this functional translation of the variables, the clocks have to be set to trigger the execution. Thereby, the way the clocks can be set is restricted by the original control flow of the program, and by the clock tree. A scheduler was introduced which gets this information from the functional part of the translation and sets the clocks accordingly.

Finally, the extension was evaluated based on some examples. Thereby, as a bigger example, it was explained how a JPEG decoder can be implemented in pure QUARTZ and how the extension can be used to describe it. The several parts of the decoder can be implemented independently based on unrelated clocks, and the parts only synchronize when data is exchanged. Even though, this seems like a classical data-flow application, the imperative synchronous language showed its advantage in using control flow to describe the actual algorithms for IDCT and Huffman decoding. Furthermore, this example also implements the first notion of constructivity: variables are not immediately written in substeps. Hence, even this simplified case of constructivity turned out to be practically usable. In addition, the hardware translation was compared based on the resulting circuits.

References

- [Ager79] T. Agerwala. **Putting Petri nets to work**. *IEEE Computer*, 12(12):85–94, December 1979.
- [AhNR74] N. Ahmed, T. Natarajan, and K.R. Rao. **Discrete cosine transform**. *IEEE Transactions on Computers*, 23(1):90–93, January 1974.
- [Alle70] F.E. Allen. **Control flow analysis**. *ACM SIGPLAN Notices*, 5(7):1–19, 1970.
- [AmBG94] T. Amagbegnon, L. Besnard, and P. Le Guernic. **Arborescent canonical form of boolean expressions**. Internal Report 826, Institut Recherche en Informatique et Systeme Aleatoire (IRISA), Rennes, France, 1994.
- [Andr03] C. André. **Computing SynCharts reactions**. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 88:3–19, 2003. Workshop on Synchronous Languages, Applications, and Programming (SLAP).
- [Andr95] C. André. **SyncCharts: A visual representation of reactive behaviors**. Research Report tr95-52, University of Nice, Sophia Antipolis, France, 1995.
- [Arvi03] Arvind. **Bluespec: A language for hardware design, simulation, synthesis and verification invited talk**. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 249–254, Mont Saint-Michel, France, 2003. IEEE Computer Society.
- [AVEREST] **The AVEREST system**. <http://www.averest.org/>. Accessed: 2013-04-16.
- [BaBS10] D. Baudisch, J. Brandt, and K. Schneider. **Multithreaded code from synchronous programs: Extracting independent threads for OpenMP**. In *Design, Automation and Test in Europe (DATE)*, pages 949–952, Dresden, Germany, 2010. EDA Consortium.
- [BaBS10a] D. Baudisch, J. Brandt, and K. Schneider. **Multithreaded code from synchronous programs: Generating software pipelines for OpenMP**. In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 11–20, Dresden, Germany, 2010. Fraunhofer Verlag.

- [BaBS11b] D. Baudisch, J. Brandt, and K. Schneider. **Translating synchronous systems to data-flow process networks**. In S.-S. Yeo, B. Vaidya, and G.A. Papadopoulos, editors, *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 354–361, Gwangju, Korea, 2011. IEEE Computer Society.
- [BaBS12] D. Baudisch, J. Brandt, and K. Schneider. **Out-of-order execution of synchronous data-flow networks**. In J. McAllister and S. Bhattacharyya, editors, *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS)*, pages 168–175, Samos, Greece, 2012. IEEE Computer Society.
- [BCEH03] 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.
- [BeBe91] A. Benveniste and G. Berry. **The synchronous approach to reactive real-time systems**. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
- [BeCo85] G. Berry and L. Cosserat. **The Esterel synchronous programming language and its mathematical semantics**. In S.D. Brookes, A.W. Roscoe, and G. Winskel, editors, *Seminar on Concurrency (CONCUR)*, volume 197 of *LNCS*, pages 389–448, Pittsburgh, Pennsylvania, USA, 1985. Springer.
- [BeHH92] G. Berry, C.A.R. Hoare, and W.A. Hunt. **Esterel on hardware**. *Philosophical Transactions of the Royal Society of London*, 339(1652):87–104, April 1992.
- [BeKi00] G. Berry and M. Kishinevsky. **Hardware Esterel language extension proposal**, 2000. Research Proposal.
- [Berr00] G. Berry. **The Esterel v5 language primer**, July 2000.
- [Berr89] G. Berry. **Real-time programming: General purpose or special-purpose languages**. *Information Processing*, pages 11–17, 1989.
- [Berr91] G. Berry. **A hardware implementation of pure Esterel**. In *Formal Methods in VLSI Design*, Miami, Florida, USA, 1991.
- [Berr92] G. Berry. **A hardware implementation of pure Esterel**. *Sadhana*, 17(1):95–130, March 1992.
- [Berr97a] G. Berry. **The Esterel v5 language primer**. <http://www.inria.fr/meije/esterel/>, April 1997.
- [Berr97b] G. Berry. **A quick guide to Esterel**, February 1997.
- [Berr99] G. Berry. **The constructive semantics of pure Esterel**, July 1999.
- [BeSe01] G. Berry and E. Sentovich. **Multiclock Esterel**. In T. Margaria and T.F. Melham, editors, *Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 110–125, Livingston, Scotland, UK, 2001. Springer.

- [BGSS11] J. Brandt, M. Gemünde, K. Schneider, S. Shukla, and J.-P. Talpin. **Integrating system descriptions by clocked guarded actions.** In A. Morawiec, J. Hinderscheit, and O. Ghenassia, editors, *Forum on Specification and Design Languages (FDL)*, pages 1–8, Oldenburg, Germany, 2011. IEEE Computer Society.
- [BGSS12] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. **Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions.** *Design Automation for Embedded Systems (DAEM)*, July 2012. DOI 10.1007/s10617-012-9087-9.
- [BGSS13] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. **Embedding polychrony into synchrony.** *IEEE Transactions on Software Engineering (TSE)*, 2013.
- [Bous98] F. Boussinot. **SugarCubes implementation of causality.** Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis, France, September 1998.
- [BrGS09] J. Brandt, M. Gemünde, and K. Schneider. **Desynchronizing synchronous programs by modes.** In S. Edwards, R. Lorenz, and W. Vogler, editors, *Application of Concurrency to System Design (ACSD)*, pages 32–41, Augsburg, Germany, 2009. IEEE Computer Society.
- [BrGS10] J. Brandt, M. Gemünde, and K. Schneider. **From synchronous guarded actions to SystemC.** In M. Dietrich, editor, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 187–196, Dresden, Germany, 2010. Fraunhofer Verlag.
- [BrSc09] J. Brandt and K. Schneider. **Separate compilation for synchronous programs.** In H. Falk, editor, *Software and Compilers for Embedded Systems (SCOPES)*, volume 320 of *ACM International Conference Proceeding Series*, pages 1–10, Nice, France, 2009. ACM.
- [BrSc11a] J. Brandt and K. Schneider. **Separate translation of synchronous programs to guarded actions.** Internal Report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, March 2011.
- [BrSe95] J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, 1995.
- [CaLa08] C.G. Cassandras and S. Lafortune. *Introduction to Discrete Event Systems*. Springer, 2 edition, 2008.
- [CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J.A. Plaice. **LUSTRE: A declarative language for programming synchronous systems.** In *Principles of Programming Languages (POPL)*, pages 178–188, Munich, Germany, 1987. ACM.
- [DiWa04] V. Dimitrov and K. Wahid. **Multiplierless DCT algorithm for image compression applications.** *International Journal on Information Theory*

- and Applications (IJ ITA)*, 11:162–169, 2004.
- [EdTa05] S.A. Edwards and O. Tardieu. **SHIM: a deterministic model for heterogeneous embedded systems**. In W. Wolf, editor, *Embedded Software (EMSOFT)*, pages 264–272, Jersey City, New Jersey, USA, 2005. ACM.
- [Edwa00] S.A. Edwards. **Compiling Esterel into sequential code**. In *Design Automation Conference (DAC)*, pages 322–327, Los Angeles, California, USA, 2000. ACM.
- [Edwa03a] S.A. Edwards. **Making cyclic circuits acyclic**. In *Design Automation Conference (DAC)*, pages 159–162, Anaheim, California, USA, 2003. ACM.
- [Edwa05b] S.A. Edwards. **SHIM: A language for hardware/software integration**. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005.
- [FeOW87] J. Ferrante, K.J. Ottenstein, and J.D. Warren. **The program dependence graph and its use in optimization**. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987.
- [FeWi92] E. Feig and S. Winograd. **Fast algorithm for the discrete cosine transform**. *IEEE Transactions on Signal Processing*, 40(9):2174–2193, September 1992.
- [GaGB87] T. Gautier, P. Le Guernic, and L. Besnard. **SIGNAL, a declarative language for synchronous programming of real-time systems**. In G. Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 257–277, Portland, Oregon, USA, 1987. Springer.
- [Gama10] A. Gamatie. *Designing Embedded Systems with the SIGNAL Programming Language*. Springer, 2010.
- [GeBS10] M. Gemünde, J. Brandt, and K. Schneider. **Clock refinement in imperative synchronous languages**. In A. Benveniste, S.A. Edwards, E. Lee, K. Schneider, and R. von Hanxleden, editors, *SYNCHRON’09: Abstracts Collection of Dagstuhl Seminar 09481*, Dagstuhl Seminar Proceedings, pages 3–21, Dagstuhl, Germany, 2010. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany. ISSN 1862-4405, <http://www.dagstuhl.de/09481>.
- [GeBS10a] M. Gemünde, J. Brandt, and K. Schneider. **A formal semantics of clock refinement in imperative synchronous languages**. In L. Gomes, V. Khomenko, and J.M. Fernandes, editors, *Application of Concurrency to System Design (ACSD)*, pages 157–168, Braga, Portugal, 2010. IEEE Computer Society.
- [GeBS10b] M. Gemünde, J. Brandt, and K. Schneider. **Compilation of imperative synchronous programs with refined clocks**. In L. Carloni and B. Jobstmann, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 209–218, Grenoble, France, 2010. IEEE Computer Society.

- [GeBS11] M. Gemünde, J. Brandt, and K. Schneider. **Schizophrenia and causality in the context of refined clocks**. In A. Morawiec, J. Hinderscheit, and O. Ghenassia, editors, *Forum on Specification and Design Languages (FDL)*, pages 1–8, Oldenburg, Germany, 2011. IEEE Computer Society.
- [GeBS11a] M. Gemünde, J. Brandt, and K. Schneider. **Causality analysis of synchronous programs with refined clocks**. In *High Level Design Validation and Test Workshop (HLDVT)*, pages 25–32, Napa, California, USA, 2011. IEEE Computer Society.
- [GeBS13] M. Gemünde, J. Brandt, and K. Schneider. **Clock refinement in imperative synchronous languages**. *EURASIP Journal on Embedded Systems*, 2013.
- [GGBM91] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. **Programming real-time applications with SIGNAL**. *Proceedings of the IEEE*, 79(9):1321–1336, 1991.
- [GiLL99] A. Girault, B. Lee, and E.A. Lee. **Hierarchical finite state machines with multiple concurrency models**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 18(6):742–760, June 1999.
- [GiNi03] A. Girault and X. Nicollin. **Clock-driven automatic distribution of Lustre programs**. In R. Alur and I. Lee, editors, *Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 206–222, Philadelphia, Pennsylvania, USA, 2003. Springer.
- [GuTL03] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann. **Polychrony for system design**. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):261–304, June 2003.
- [Halb93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
- [HaMa95] N. Halbwachs and F. Maraninchi. **On the symbolic analysis of combinatorial loops in circuits and synchronous programs**. In *Euromicro Conference*, Como, Italy, 1995. IEEE Computer Society.
- [HaPn85] D. Harel and A. Pnueli. **On the development of reactive systems**. In K.R. Apt, editor, *Logic and Models of Concurrent Systems*, pages 477–498. Springer, 1985.
- [Hare87] D. Harel. **Statecharts: A visual formulation for complex systems**. *Science of Computer Programming*, 8(3):231–274, 1987.
- [HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. **The synchronous dataflow programming language LUSTRE**. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
- [HMAD13] R. von Hanxleden, M. Mendler, J. Aguado, B. Duderstadt, I. Fuhrmann, C. Motika, S. Mercer, and O. O’Brien. **Sequentially constructive concurrency: a conservative extension of the synchronous model of com-**

- putation.** In E. Macii, editor, *Design, Automation and Test in Europe (DATE)*, pages 581–586, Grenoble, France, 2013. EDA Consortium/ACM.
- [Hoar78] C.A.R. Hoare. **Communicating sequential processes.** *Communications of the ACM (CACM)*, 21(8):666–677, 1978.
- [Hoar83] C.A.R. Hoare. **Communicating sequential processes.** *Communications of the ACM (CACM)*, 26(1):100–106, January 1983.
- [HoAr99] J.C. Hoe and Arvind. **Hardware synthesis from term rewriting systems.** Technical Report CSG-MEMO 421-a, Computer Science and Artificial Intelligence Laboratory, Cambridge, Massachusetts, USA, 1999.
- [Huff52] D.A. Huffman. **A method for the construction of minimum-redundancy codes.** *Proceedings of the IRE*, 40(9):1098–1101, 1952.
- [IEEE05] IEEE. *IEEE Standard SystemC Language Reference Manual.* New York, New York, USA, December 2005. IEEE Std. 1666-2005.
- [IEEE05a] IEEE. *IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language*, 2005. IEEE Std. 1394-2005.
- [IEEE08] IEEE. *IEEE Standard VHDL Language Reference Manual*, 2008. IEEE Std. 1076-2008.
- [Jant04] A. Jantsch. *Modeling Embedded Systems and SoCs.* Morgan Kaufmann, 2004.
- [JFIF] E. Hamilton. **Jpeg file interchange format.** <http://www.jpeg.org/public/jfif.pdf>, September 1992. Accessed: 2013-04-29.
- [JoPS10] B.A. Jose, J. Pribble, and S.K. Shukla. **Faster software synthesis using actor elimination techniques for polychronous formalism.** In *Application of Concurrency to System Design (ACSD)*, pages 147–156, Braga, Portugal, 2010. IEEE Computer Society.
- [JoSh10] B.A. Jose and S.K. Shukla. **An alternative polychronous model and synthesis methodology for model-driven embedded software.** In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 13–18, Taipei, China, 2010. IEEE Computer Society.
- [JPSS09] B.A. Jose, J. Pribble, L. Stewart, and S.K. Shukla. **EmCodeSyn: A visual framework for multi-rate data flow specifications and code synthesis for embedded applications.** In *Forum on Specification and Design Languages (FDL)*, pages 1–6, Sophia Antipolis, France, 2009. IEEE Computer Society.
- [Kahn74] G. Kahn. **The semantics of a simple language for parallel programming.** In J.L. Rosenfeld, editor, *Information Processing*, pages 471–475, Stockholm, Sweden, 1974. North-Holland.
- [Lee06] E.A. Lee. **The problem with threads.** *IEEE Computer*, 39(5):33–42, 2006.

- [LeMe87] E.A. Lee and D.G. Messerschmitt. **Static scheduling of synchronous data flow programs for digital signal processing**. *IEEE Transactions on Computers (T-C)*, 36(1):24–35, January 1987.
- [LeMe87a] E.A. Lee and D.G. Messerschmitt. **Synchronous data flow**. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.
- [LeSa98] E.A. Lee and A. Sangiovanni-Vincentelli. **A framework for comparing models of computation**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 17(12):1217–1229, December 1998.
- [LLBH05] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. von Hanxleden. **An Esterel processor with full preemption support and its worst case reaction time analysis**. In T.M. Conte, P. Faraboschi, W.H. Mangione-Smith, and W.A. Najjar, editors, *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 225–236, San Francisco, California, USA, 2005. ACM.
- [LoLM89] C. Loeffler, A. Ligtenberg, and G.S. Moschytz. **Practical fast 1-D DCT algorithms with 11 multiplications**. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 988–991, Glasgow, Scotland, UK, 1989. IEEE Computer Society.
- [LUSv6] **The lustre v6 reference manual (draft)**. <http://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf>, January 2013.
- [Mali94] S. Malik. **Analysis of cycle combinational circuits**. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 13(7):950–956, July 1994.
- [MeHT09] M. Mendler, R. von Hanxleden, and C. Traulsen. **WCRT algebra and interfaces for Esterel-style synchronous processing**. In *Design, Automation and Test in Europe (DATE)*, pages 93–98, Nice, France, 2009. IEEE Computer Society.
- [Moss06] P.D. Mosses. **Formal semantics of programming languages**. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 148:41–73, 2006.
- [Nebu03] M. Nebut. **An overview of the Signal clock calculus**. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 88:39–54, 2003. Workshop on Synchronous Languages, Applications, and Programming (SLAP).
- [OtOt84] K.J. Ottenstein and L.M. Ottenstein. **The program dependence graph in a software development environment**. In *Software Engineering Symposium on Practical Software Development Environments*, pages 177–184. ACM, 1984.
- [Petr80a] C.A. Petri. **Introduction to general net theory**. In W. Brauer, editor, *Net Theory and Applications*, volume 84 of *LNCS*, pages 1–19, Hamburg, Germany, 1980. Springer.

- [Plot81] G.D. Plotkin. **A structural approach to operational semantics**. Technical Report FN-19, DAIMI, Århus, Denmark, 1981.
- [PoST05] D. Potop-Butucaru, R. de Simone, and J.-P. Talpin. **The synchronous hypothesis and synchronous languages**. In R. Zurawski, editor, *Embedded Systems Handbook*, volume 2 of *Industrial Information Technology*, chapter 8. CRC Press, 2005.
- [PrTH06] S. Prochnow, C. Traulsen, and R. von Hanxleden. **Synthesizing safe state machines from Esterel**. In M.J. Irwin and K. De Bosschere, editors, *Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 113–124, Ottawa, Ontario, Canada, 2006. ACM.
- [RaSh00] B. Rajan and R.K. Shyamasundar. **Modeling VHDL in multiclock ESTEREL**. In *VLSI Design (VLSID)*, pages 76–83, Calcutta, India, 2000. IEEE Computer Society.
- [RaSh00a] B. Rajan and R.K. Shyamasundar. **Multiclock ESTEREL: A reactive framework for asynchronous design**. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pages 201–209, Cancún, Quintana Roo, Mexico, 2000. IEEE Computer Society.
- [RaSh00b] B. Rajan and R.K. Shyamasundar. **Modeling distributed embedded systems in multiclock Esterel**. In T. Bolognesi and D. Latella, editors, *Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE/PSTV)*, pages 301–316, Pisa, Italy, 2000. Kluwer.
- [RoSa08a] C. Rousseau and Y. Saint-Aubin. **Image compression the jpeg standard**. In *Mathematics and Technology*, Springer Undergraduate Texts in Mathematics and Technology, pages 1–33. Springer New York, 2008.
- [RSAS07] S.S. Ravi, G. Singh, S. Ahuja, and S.K. Shukla. **Complexity of scheduling in synthesizing hardware from concurrent action oriented specifications**. In L. Benini, N. Chang, U. Kremer, and C.W. Probst, editors, *Power-Aware Computing Systems*, volume 07041 of *Dagstuhl Seminar Proceedings*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI) Schloss Dagstuhl. <http://drops.dagstuhl.de/opus/volltexte/2007/1105>.
- [SaMo10] D. Salomon and G. Motta. *Handbook of Data Compression*. Springer London, 2010.
- [SBST05b] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. **Maximal causality analysis**. In J. Desel and Y. Watanabe, editors, *Application of Concurrency to System Design (ACSD)*, pages 106–115, Saint-Malo, France, 2005. IEEE Computer Society.
- [ScBr08] K. Schneider and J. Brandt. **Performing causality analysis by bounded model checking**. In *Application of Concurrency to System Design (ACSD)*, pages 78–87, Xi’an, China, 2008. IEEE Computer Society.

- [ScBS04b] K. Schneider, J. Brandt, and T. Schuele. **Causality analysis of synchronous programs with delayed actions.** In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, District of Columbia, USA, 2004. ACM.
- [ScBS06] K. Schneider, J. Brandt, and T. Schuele. **A verified compiler for synchronous programs with local declarations.** *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.
- [Schn00] K. Schneider. **A verified hardware synthesis for Esterel.** In F.J. Rammig, editor, *Distributed and Parallel Embedded Systems (DIPES)*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer.
- [Schn01a] K. Schneider. **Embedding imperative synchronous languages in interactive theorem provers.** In *Application of Concurrency to System Design (ACSD)*, pages 143–154, Newcastle Upon Tyne, England, UK, 2001. IEEE Computer Society.
- [Schn09] K. Schneider. **The synchronous programming language Quartz.** Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.
- [ScWe01] K. Schneider and M. Wenz. **A new method for compiling schizophrenic synchronous programs.** In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, 2001. ACM.
- [ShBT96] T.R. Shiple, G. Berry, and H. Touati. **Constructive analysis of cyclic circuits.** In *European Design Automation Conference (EDAC)*, pages 328–333, Paris, France, 1996. IEEE Computer Society.
- [ShTa10] S.K. Shukla and J.-P. Talpin. *Synthesis of Embedded Software – Frameworks and Methodologies for Correctness by Construction.* Springer, 2010.
- [SiSh07] G. Singh and S.K. Shukla. **Algorithms for low power hardware synthesis from concurrent action oriented specifications CAOS.** *International Journal of Embedded Systems (IJES)*, 3(1/2):83–92, 2007.
- [SSBD99] S.A. Seshia, R.K. Shyamasundar, A.K. Bhattacharjee, and S.D. Dhodapkar. **A translation of Statecharts to Esterel.** In J.M. Wing, J. Woodcock, and J. Davies, editors, *Formal Methods (FM)*, volume 1709 of *LNCS*, pages 983–1007, Toulouse, France, 1999. Springer.
- [TaSi04] O. Tardieu and R. de Simone. **Curing schizophrenia by program rewriting in Esterel.** In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 39–48, San Diego, California, USA, 2004. IEEE Computer Society.
- [TBGS13] J.-P. Talpin, J. Brandt, M. Gemünde, K. Schneider, and S. Shukla. **Constructive polychronous systems.** In S.N. Artëmov and A. Nerode, editors, *Logical Foundations of Computer Science (LFCS)*, volume 7734 of *LNCS*, pages 335–349, San Diego, California, USA, 2013. Springer.

- [Tini00] S. Tini. *Structural Operational Semantics for Synchronous Languages*. PhD thesis, University of Pisa, Italy, 2000.
- [Wall91] G.K. Wallace. **The JPEG still picture compression standard**. *Communications of the ACM (CACM)*, 34(4):30–44, April 1991.
- [YKSH09] J.-H. Yun, C.-J. Kim, S. Seo, T. Han, and K.-M. Choe. **Refining schizophrenia via graph reachability in Esterel**. In R. Bloem and P. Schaumont, editors, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 18–27, Cambridge, Massachusetts, USA, 2009. IEEE Computer Society.

Curriculum Vitae

Persönliche Daten

Name	Mike Gemünde
Geburtstag / -ort	26.04.1983 / Bad Kreuznach
Familienstand	ledig
Staatsangehörigkeit	deutsch

Schulbildung

1989 – 1993	Grundschule Sprendlingen
1993 – 1999	Realschule Wörrstadt
1999 – 2002	Technisches Gymnasium in Mainz Abschluss: Abitur

Wehrdienst

10/2002 – 07/2003	Grundwehrdienst, Spezialpionierbataillon 464 Speyer
--------------------------	---

Hochschulstudium

10/2003 – 07/2008	Technische Universität Kaiserslautern Fachbereich Informatik Abschluss: Diplom-Informatiker (Dipl.-Inf.)
--------------------------	--

Berufserfahrung

09/2008 – 08/2013	Technische Universität Kaiserslautern Wissenschaftlicher Mitarbeiter, Fachbereich Informatik
seit 09/2013	Continental AG in Frankfurt

