2 Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems

Klaus Schneider and Jens Brandt

Abstract

Since the synchronous model of computation is shared between synchronous languages and synchronous hardware circuits, synchronous languages lend themselves well for hardware/software codesign in the sense that from the same synchronous program both hardware and software can be generated. In this chapter, we informally describe the syntax and semantics of the imperative synchronous language Quartz and explain how these programs are first analyzed and then compiled to hardware and software: To this end, the programs are translated to synchronous guarded actions whose causality has to be ensured as a major consistency analysis of the compiler. We then explain the synthesis of hardware circuits and sequential programs from synchronous guarded actions and briefly look at extensions of the Quartz language in the conclusions.

Acronyms

AIF	Averest Intermediate Format
EFSM	Extended Finite-State Machine
MoC	Model of Computation
SMV	Symbolic Model Verifier
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

K. Schneider (✉)
Embedded Systems Group, University of Kaiserslautern, Kaiserslautern, Germany
e-mail: schneider@cs.uni-kl.de

J. Brandt
Faculty of Electrical Engineering and Computer Science, Hochschule Niederrhein, Krefeld, Germany
e-mail: jens.brandt@hsnr.de

© Springer Science+Business Media Dordrecht 2017

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_3

Contents

2.1 Introduction . . . 30
2.2 The Synchronous Language Quartz . . . 31
2.3 Compilation . . . 35
2.3.1 Intermediate Representation by Guarded Actions . . . 35
2.3.2 Surface and Depth . . . 37
2.3.3 Compilation of the Control Flow . . . 38
2.3.4 Compilation of the Data Flow . . . 41
2.3.5 Local Variables and Schizophrenia . . . 42
2.4 Semantic Analysis . . . 45
2.5 Synthesis . . . 46
2.5.1 Symbolic Model Checking . . . 47
2.5.2 Circuit Synthesis . . . 48
2.5.3 SystemC Simulation . . . 50
2.5.4 Automaton-Based Sequential Software Synthesis . . . 51
2.6 Conclusions and Future Extensions . . . 55
References . . . 56

2.1 Introduction

Compared to traditional software development, the design of embedded systems is even more challenging: In addition to the correct implementation of the functional behavior, one has to consider also non-functional constraints such as real-time behavior, reliability, and energy consumption. For this reason, embedded systems are often built with specialized, often application-specific hardware platforms. To allow late design changes even on the hardware/software partitioning, languages and model-based design tools are required that can generate both hardware and software from the same realization-independent model. Moreover, many embedded systems are used in safety-critical applications where errors can lead to severe damages up to the loss of human lives. For this reason, formal verification is applied in many design flows using different kinds of formal verification methods.

The *synchronous Model of Computation (MoC)* [2] has shown to be well-suited to provide realization-independent models for a model-based design of embedded reactive systems where both hardware and software can be generated. There are at least the following reasons for this success: (1) It is possible to determine tight bounds on the reaction time by a simplified worst-case execution time analysis [28, 29, 31, 32, 50], since by construction of the programs, only a statically bounded finite number of actions can be executed within each reaction step. (2) The formal semantics of these languages allows one to prove (a) the correctness of the compilation and (b) the correctness of particular programs with respect to given formal specifications [40,42,44,47]. (3) It is possible to generate both efficient software and hardware from the same synchronous programs. Since the synchronous MoC is also used by synchronous hardware circuits, the translation to synchronous hardware circuits [3,4,38–40,47] is conceptually clearer and simpler than for classic hardware description languages such as VHSIC Hardware Description Language (VHDL) or Verilog (at least if these are not restricted to synchronous synthesizable subsets).

All these advantages are due to the *synchronous MoC* that postulates two essential properties: (1) A run of a synchronous system consists of a linear sequence of discrete reactions. In each of these reactions, a synchronous program reads the inputs, updates the internal state, and computes the values of the corresponding outputs. (2) All the computations within such a reaction are virtually performed in zero time, i.e., they are supposed to happen at the very same point of time so that every computation can immediately see the effects of every other computation within the same reaction step. Of course, this is not possible in a real system, but executing all computations according to the underlying data dependencies gives the programmer the impression postulated by the synchronous MoC.

The synchronous MoC simplifies the design of reactive embedded systems, since developers do not have to care about low-level details like timing, synchronization, and scheduling. Instead, the synchronous paradigm poses some specific problems to compilers, in particular, the *causality analysis* [4, 7, 13, 24, 34, 45, 48, 52] of synchronous programs. Intuitively, causality cycles occur when the input of a computation depends on its own output. Causally correct programs therefore have a causal order of the actions in every macro step, which allows the program to compute values before they are read. Another important analysis is the *clock consistency* that is required for synchronous programs with more than one clock: Here, we have to ensure that variables are only read at points of time when they are defined and that they are only assigned values whenever their clocks allow this. Although these and other problems turned out to be quite challenging, research over the last three decades has considered these problems in detail, found practical algorithms, and developed various compilers like [6, 11, 16, 17, 19, 21, 36, 37, 47] that are able to generate both hardware and software from one and the same synchronous program.

This chapter gives an overview of our synchronous language Quartz, which was derived from the pioneering language Esterel with a special focus on hardware design and formal verification. We start with a presentation of the language statements in Sect. 2.2 and explain with a few illustrative examples how it adheres with the synchronous MoC. The subsequent sections give more details of our Quartz compiler as implemented in our Averest system (http://www.averest.org): Sect. 2.3 shows how the compiler translates the source code into an intermediate representation which is subsequently used as the starting point for analysis and synthesis. Section 2.4 explains how causality is analyzed, before Sect. 2.5 describes how the intermediate representation is finally transformed to executable software or hardware. Finally, we briefly have a look at planned extensions of the language in the conclusions in Sect. 2.6.

2.2 The Synchronous Language Quartz

As outlined in the introduction, the synchronous MoC assumes that the execution consists of a sequence of reactions $\mathscr{R} = \langle R_0, R_1, \ldots \rangle$. In each reaction (also called macro step [25]), all the actions (also called micro steps) that take place

within a reaction are executed according to their data dependencies. This leads to the programmer's view that the execution of micro steps does not take time and that every macro step of a synchronous program requires the same amount of logical time. As a consequence, concurrent threads run in lockstep and automatically synchronize at the end of their macro steps, which yields the very important fact that even concurrent synchronous programs still have deterministic behaviors (indeed, most concurrent programming models lead to nondeterministic behaviors). As a result, *deterministic* single-threaded code can be obtained from multi-threaded synchronous programs. Thus, software generated from synchronous programs can be executed on ordinary microcontrollers without having the need of complex process scheduling of operating systems.

Synchronous hardware circuits are well-known models that are also based on the synchronous MoC: Here, each reaction is initiated by a clock signal, and all parts of the circuits are activated simultaneously. Although the signals need time to pass gates (computation) and wires (communication), propagation delays can be safely neglected as long as signals stabilize before the next clock tick arrives. Variables are either mapped to wires or registers, which both have unique values for every cycle. All these correspondences make the modeling and synthesis of synchronous circuits from synchronous programs very appealing [3, 4, 38–40, 47]. It is therefore not surprising that causality analysis of synchronous programs is a descendant of ternary simulation of asynchronous circuits [13, 33, 34, 52] and can this way be viewed as the proof of the abstraction from physical time to the abstract logical time (clocks).

In this section, we introduce our synchronous language Quartz [43], which provides the synchronous principle described above in the form of an imperative programming language similar to its forerunner Esterel [5, 6]. In the following, we give a brief overview of the core of the language that is sufficient to define most other statements as simple syntactic sugar. For the sake of simplicity, we do not give a formal definition of the semantics; the interested reader is referred to [43], which also provides a complete structural operational semantics in full detail. The Quartz core consists of the statements listed in Fig. 2.1, provided that S, S_1, and S_2 are also

`nothing`	(empty statement)
ℓ : `pause`	(start/end of macro step)
$x = \tau$ and `next`$(x) = \tau$	(assignments)
`if`(σ) S_1 `else` S_2	(conditional)
$S_1 ; S_2$	(sequence)
`do` S `while`(σ)	(iteration)
$S_1 \parallel S_2$	(synchronous concurrency)
`[weak] [immediate] abort` S `when`(σ)	(preemption: abortion)
`[weak] [immediate] suspend` S `when`(σ)	(preemption: suspension)
$\{\alpha\ x;\ S\}$	(local variable x of type α)
$inst : name(\tau_1, \ldots, \tau_n)$	(call of module *name*)

Fig. 2.1 The Quartz core statements

core statements, ℓ is a label, x and τ are a variable and an expression of the same type, σ is a Boolean expression, and α is a type.

Values of variables (or often called signals in the context of synchronous languages) of the synchronous program can be modified by assignments. They immediately evaluate the right-hand side expression τ in the current environment/macro step. Immediate assignments $x = \tau$ instantaneously transfer the obtained value of τ to the left-hand side x in the current macro step, whereas delayed assignments $\texttt{next}(x) = \tau$ transfer this value only in the following macro step. For this reason, all micro step actions are evaluated in the same variable environment which is also determined by these actions. The causality analysis makes sure that this cyclic dependency can be constructively resolved in a deterministic way.

The synchronous programming paradigm is therefore different to traditional sequential programs: For example, the incrementation of a loop variable i by an assignment $i = i + 1$ does not make sense in synchronous languages, since this requires to compute a solution to the (unsolvable) equation $i = i + 1$. Using delayed actions, one can write $\mathbf{next}(i) = i + 1$, which is the true intention of the assignment. Even more difficult is the interaction of several micro step actions, e.g., in the same sequence or in different parallel substatements: Their order given in the program does not matter for the execution, since execution just follows the data dependencies in a read-after-write schedule: In legal schedules, variables are only read if their value for the current macro step has already been determined. For example, the program in Fig. 2.2a has the same behavior as the program in Fig. 2.2b. Thus, every statement knows and depends on the results of all operations in the current macro step. In particular, a Quartz statement may influence its own activation condition (see the program in Fig. 2.2c). Obviously, this generally leads to causal cycles that have to be analyzed by the compilers, i.e., the compilers have to ensure that for all inputs, there is an execution order of the micro step actions such that a variable is never read before it is written in the macro step.

If a variable's value is not determined by an action in the current macro step, its value is determined by the so-called *reaction to absence*, which depends on the *storage type* of the variable. The current version of Quartz knows two of them: *memorized variables* keep the value of the previous step, while *event* variables are reset to a default value if no action determines their values. Future versions of Quartz will contain further storage types for hybrid systems and multi-clocked synchronous systems.

a	b	c
$a = 1$;	$b = a$;	$a = 1$;
$b = a$;	$a = 1$;	$\texttt{if}(b = 1)\ b = a$;
pause;	pause;	pause;
$a = b$;	$a = b$;	$\texttt{if}(a \neq 2)\ a = b$;

Fig. 2.2 Three Quartz programs illustrating the synchronous MoC

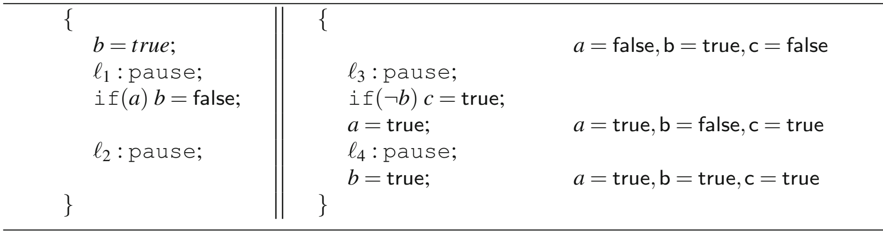

Fig. 2.3 Synchronous concurrency in Quartz

In addition to the usual control flow statements known from typical imperative languages (conditionals, sequences, and iterations), Quartz also offers synchronous concurrency. The *parallel statement* $S_1 \| S_2$ immediately starts the statements S_1 and S_2. Then, both S_1 and S_2 run in lockstep, i.e., they automatically synchronize when they reach their next `pause` statements. The parallel statement runs as long as at least one of the substatements is active.

Figure 2.3 shows a simple example consisting of two parallel threads with Boolean memorized variables a, b, and c. The threads are shown on the left-hand side of Fig. 2.3, and their effects are shown on the right-hand side. Initially, the default values of the variables are false, but in the first step, these values can be changed by the program. If the program is started, both threads are started. The first thread executes the assignment to b and stops at location ℓ_1, while the second thread immediately stops at location ℓ_3. In the second macro step, the program resumes from the labels ℓ_1 and ℓ_3. Thereby, the first thread cannot yet proceed since the value of a in this step is not yet known. The second thread cannot execute its if-statement either, but it can execute the following assignment to a. Thus, the second thread assigns true to a, then the first thread can assign false to b, and then the second thread can finally assign true to c. The last step then resumes from ℓ_2 and ℓ_4, where the second thread performs the final assignment to variable b.

Preemption of system behaviors is very important for reactive systems. It is therefore very convenient that languages like Esterel and Quartz offer `abort` and `suspend` statements to explicitly express preemption. The meaning of these statements is as follows: A statement S which is enclosed by an `abort` block is immediately terminated when the given condition σ holds. Similarly, the `suspend` statement freezes the control flow in a statement S when σ holds. Thereby, two kinds of preemption must be distinguished: strong (default) and weak (indicated by keyword `weak`) preemption. While strong preemption deactivates both the control and data flow of the current step, weak preemption only deactivates the control flow, but retains the data flow of the current macro step (this concept is sometimes also called ‘run-to-completion’). The immediate variants check for preemption already at starting time, while the default is to check preemption only after starting time.

Modular design is supported by the declaration of modules in the source code and by calling these modules in statements. Any statement can be encapsulated in a

module, which further declares a set of input and output signals for interaction with its context statement. There are no restrictions for module calls, so that modules can be instantiated in every statement. In contrast to many other languages, a module instantiation can also be part of sequences or conditionals (and is therefore not restricted to be called as additional thread). Furthermore, it can be located in any abortion or suspension context, which possibly preempts its execution.

The pioneer of the class of imperative synchronous languages is Esterel. As the description above suggests, both Esterel and Quartz share many principles, but there are also some subtle differences: First, the concept of pure and valued signals in Esterel has been generalized to *event* and *memorized* variables in Quartz as presented above. In particular, there is no distinction between `if` and `present` statements, and the reaction to absence considers arbitrary variables and not just signals. Another important difference is causality: while Esterel usually sees the statements $S_1; S_2$ as a real sequence, where the backward flow of information from S_2 to S_1 is forbidden, Quartz has a more relaxed definition of that. As long as S_1 and S_2 are executed in the same macro step, information can be arbitrarily exchanged, as the example in Fig. 2.3 illustrates. In consequence, more programs are considered to be causally correct. In contrast to Esterel, Quartz offers also the delayed assignments $\texttt{next}(x) = \tau$ that are convenient to describe hardware designs (while Esterel makes use of a `pre` operator to refer to the previous value of a signal).

2.3 Compilation

2.3.1 Intermediate Representation by Guarded Actions

Having presented the syntax and semantics of the synchronous language Quartz in the previous section, we now describe how it is compiled to hardware and software systems. As usual for compilers, we thereby make use of an internal representation of the program that can be used for analysis and the later synthesis. Especially in the design of embedded systems, where hardware-software partitioning and target platforms are design decisions that are frequently changed, persistent intermediate results stored in an internal representation are very important. It is thereby natural to distinguish between *compilation and synthesis*: Compilation is the translation of the Quartz program into the internal representation, and synthesis is the translation from the internal representation to traditional hardware or software descriptions.

In our Averest system, which is a framework for the synthesis and verification of Quartz programs, we have chosen *synchronous guarded actions* as internal representation that we called Averest Intermediate Format (AIF). Synchronous guarded actions are in the spirit of traditional guarded commands [15, 18, 27, 30] but follow the synchronous MoC. The Boolean condition γ of a guarded action $\langle\gamma \Rightarrow \mathscr{C}\rangle$ is called the guard and the atomic statement $\mathscr{C}$ is called the action of the guarded action. According to the previous section, atomic statements are

essentially the assignments of Quartz, i.e., the guarded actions have either the form $\gamma \Rightarrow x = \tau$ (for an immediate assignment) or $\gamma \Rightarrow \mathsf{next}(x) = \tau$ (for a delayed assignment).

The intuition behind such a guarded action $\gamma \Rightarrow \mathscr{C}$ is that the action $\mathscr{C}$ is executed in every macro step where the condition γ is satisfied. Guarded actions may be viewed as a simple programming language like Unity [15] in that every guarded action runs as a separate process in parallel to the other guarded actions. This process observes its guard γ in each step and executes $\mathscr{C}$ if the guard holds. The *semantics of synchronous guarded actions* is simply defined as follows: In every macro step, all guards are simultaneously checked. If a guard is true, its action is immediately executed: immediate assignments instantaneously transfer the computed value to the left-hand side of the assignment, while the delayed assignments defer the transfer to the next macro step. As there may be interdependencies between actions and trigger conditions, the actions must be executed according to their data dependencies. Similar to the Quartz program, the AIF description handles the *reaction to absence* implicitly: If no action has determined the value of the variable in the current macro step (obviously, this is the case iff 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 false), then its value is determined by the reaction to absence according to its storage mode: *Event* variables are reset to their default values (like wires in hardware circuits), while *memorized* variables store their previous values (like registers in hardware circuits). Future versions of Quartz will contain clocked variables that are absent if not explicitly assigned in a step, i.e., the reaction to absence will not provide any value for them.

We are convinced that this representation of the behavior is exactly at *the right level of abstraction* for an intermediate code format, since guarded actions provide a good balance between (1) removal of complexity from the source code level and (2) the independence of a specific synthesis target. The semantics of complex control flow statements can be completely encoded by the guarded actions, so that the subsequent analysis, optimization, and synthesis steps become much simpler: Due to their very simple structure, efficient translation to both software and hardware is efficiently possible from guarded actions.

In general, programs written in all synchronous languages can be translated to synchronous guarded actions [8–10]. While this translation is straightforward for data flow languages such as Lustre, more effort is needed for imperative languages such as Quartz and Esterel. There, the translation has to extract all actions of the program and to compute for each of them a trigger condition according to the program. In the rest of this section, we describe the basics of the translation from Quartz programs to guarded actions. For a better understanding, we neglect local variables and module calls and rather describe a simple, but incomplete translation in this section. More details of the translation are given in [11].

In the following, we first discuss the distinction of surface and depth of a program in Sect. 2.3.2. Based on this distinction, we present the compilation of the control flow in Sect. 2.3.3, before we focus on the data flow in Sect. 2.3.4. Finally, we consider the additional problems due to local variables in Sect. 2.3.5.

2.3.2 Surface and Depth

A key to the compilation of synchronous programs is the distinction between the *surface and depth* of a program: Intuitively, the surface consists of the micro steps that are executed when the program is started, i.e., all the parts that are executed before reaching the first `pause` statements. The depth contains the statements that are executed when the program resumes execution after the first macro step, i.e., when the control is already inside the program and proceeds with its execution. It is important to note that surface and depth may overlap, since `pause` statements may be conditionally executed. Consider the example shown in Fig. 2.4a: while the action $x = 1$ is only in the surface and the action $z = 1$ is only in the depth, the action $y = 1$ is both in the depth (Fig. 2.4b) and the surface (Fig. 2.4c) of the sequence.

The example shown in Fig. 2.5 illustrates the necessity of distinguishing between the surface and the depth for the compilation. The compilation should compute for the data flow of a statement S guarded actions of the form $\langle \gamma \Rightarrow \mathscr{C} \rangle$, where $\mathscr{C}$ is a Quartz assignment, which is executed if and only if the condition γ holds. One may think that the set of guarded actions for the data flow can be computed by a simple recursive traversal over the program structure, which keeps track of the precondition leading to the current position. However, this is not the case, as the example in Fig. 2.5 illustrates. Since the abortion is not an immediate one, the assignment $a = \mathsf{true}$ will never be aborted, while the assignment $b = \mathsf{true}$ will be aborted if i holds. Now, assume we would first compute guarded actions for the body of the abortion statement and would then replace each guard φ by $\varphi \wedge \neg i$ to implement the abortion. For the variable a, this incorrect approach would derive two

(a)	(b)	(c)
$x = 1;$	$x = 1;$	$x = 1;$
`if`(a)	`if`(true)	`if`(false)
`pause;`	`pause;`	`pause;`
$y = 1;$	$y = 1;$	$y = 1;$
`pause;`	`pause;`	`pause;`
$z = 1;$	$z = 1;$	$z = 1;$

Fig. 2.4 Overlapping surface and depth: (**a**) Source code. (**b**) Case $a = \mathsf{true}$. (**c**) Case $a = \mathsf{false}$

```
ℓ0 : pause; do
    abort {
        a = true;
        ℓ1 : pause;
        b = true;
    } when(i);
while(true) ;
```

$\ell_0 \vee \ell_1 \Rightarrow a = \mathsf{true}$
$\ell_1 \wedge \neg i \Rightarrow b = \mathsf{true}$

Fig. 2.5 Using surface and depth for the compilation

guarded actions $\ell_0 \wedge \neg i \Rightarrow a = \mathsf{true}$ and $\ell_1 \wedge \neg i \Rightarrow a = \mathsf{true}$. However, this is obviously wrong since now both assignments $a = \mathsf{true}$ and $b = \mathsf{true}$ are aborted which is not the semantics of the program.

The example shows that we have to distinguish between the guarded actions of the surface and the depth of a statement since these must be treated differently by the preemption statements. If we store these actions in two different sets, then we can simply add the conjunct $\neg\sigma$ to the guards of the actions of the depth, while leaving the guards of the actions of the surface unchanged. For this reason, we have to compute guarded actions for the surface and the depth in two different sets.

2.3.3 Compilation of the Control Flow

The control flow of a synchronous program may only rest at its control flow locations. Hence, it is sufficient to describe all situations where the control flow can move from the set of currently active locations to the set of locations that are active at the next point of time. The control flow can therefore be described by actions of the form $\langle \gamma \Rightarrow \mathsf{next}(\ell) = \mathsf{true} \rangle$, where ℓ is a Boolean event variable modeling the control flow location and γ is a condition that is responsible for moving the control flow at the next point of time to location ℓ. Since a location is represented by an event variable, its reaction to absence resets it to false whenever no guarded action explicitly sets it.

Thus, the compiler has to extract from the synchronous program for every label ℓ a set of trigger conditions $\phi_1^\ell, \ldots, \phi_n^\ell$ to determine whether this label ℓ has to be set in the following step. Then, these conditions can be encoded as guarded actions $\langle \phi_1^\ell \Rightarrow \mathsf{next}(\ell) = \mathsf{true} \rangle, \ldots, \langle \phi_n^\ell \Rightarrow \mathsf{next}(\ell) = \mathsf{true} \rangle$. The whole control flow is then just the union of all sets for all labels. Hence, in the following, we describe how to determine the activation conditions ϕ_i^ℓ for each label ℓ of the program.

The compilation is implemented as a bottom-up procedure that extracts the control flow by a recursive traversal over the program structure: For example, for a loop, we first compile the loop body and then add the loop behavior. While descending in the recursion, we determine the following conditions and forward them to the compilation of the substatements of a given statement S:

- $\mathsf{strt}(S)$ is the current activation condition. It holds iff S is started in the current macro step.
- $\mathsf{abrt}(S)$ is the disjunction of the guards of all abort blocks which contain S. Hence, the condition holds iff S should be currently aborted.
- $\mathsf{susp}(S)$ similarly describes the suspension context: if the predicate holds, S will be suspended. Thereby, $\mathsf{abrt}(S)$ has a higher priority, i.e., if both $\mathsf{abrt}(S)$ and $\mathsf{susp}(S)$ hold, then the abortion takes place.

The compilation of S returns the following control flow predicates [41], which are used for the compilation of the surrounding compound statement.

- $\mathsf{inst}(S)$ holds iff the execution of S is currently instantaneous. This condition depends on inputs so that we compute an expression $\mathsf{inst}(S)$ depending on the current values of input, local, and output variables. In general, $\mathsf{inst}(S)$ cannot depend on the locations of S since it is checked whether the control flows through S without being caught in S. Hence, it is assumed that S is currently not active.
- $\mathsf{insd}(S)$ is the disjunction of the labels in statement S. Therefore, $\mathsf{insd}(S)$ holds at some point of time iff the control flow is currently at some location inside S, i.e., if S is active. Thus, instantaneous statements are never active, since the control flow cannot rest anywhere inside.
- $\mathsf{term}(S)$ describes all conditions where the control flow is currently somewhere inside S and wants to leave S voluntarily. Note, however, that the control flow might still be in S at the next point of time, since S may be (re)entered at the same time, e.g., by a surrounding loop statement. The expression $\mathsf{term}(S)$ therefore depends on input, local, output, and location variables. $\mathsf{term}(S)$ is false whenever the control flow is currently not inside S. In particular, $\mathsf{term}(S)$ is false for the instantaneous atomic statements.

The control flow predicates refer either to the surface or to the depth of a statement. As it will be obvious in the following, the surface uses $\mathsf{strt}(S)$ and $\mathsf{inst}(S)$, while the depth depends on $\mathsf{abrt}(S)$, $\mathsf{susp}(S)$, $\mathsf{insd}(S)$ and $\mathsf{term}(S)$. Hence, we can divide the compilation of each statement into two functions: one compiles its surface and the other one compiles its depth.

After these introductory explanations, we can now present the general structure of the compilation algorithm (see Fig. 2.6). The compilation of a system consisting of a statement S is initially started by the function $\mathsf{ControlFlow}(\mathrm{st}, S)$, which splits the task into surface and depth parts. Abort and suspend conditions for the depth are initially set to false, since there is no preemption context at this stage.

It remains to show how the surface and the depth of each statement are compiled. Thereby, we forward the previously determined control flow context for a statement S by the Boolean values $\mathrm{st} = \mathsf{strt}(S)$, $\mathrm{ab} = \mathsf{abrt}(S)$, and $\mathrm{sp} = \mathsf{susp}(S)$, while the result contains the values of the predicates $I = \mathsf{inst}(S)$, $A = \mathsf{insd}(S)$, and $T = \mathsf{term}(S)$.

Let us start with the assignments of the program. Since they do not contribute to the control flow, no guarded actions are derived from them. The computation of the control flow predicates is also very simple: An action $\mathscr{C}$ is always instantaneous $\mathsf{inst}(\mathscr{C}) = \mathsf{true}$, never active ($\mathsf{insd}(\mathscr{C}) = \mathsf{false}$), and never terminates (since the control flow cannot rest inside $\mathscr{C}$; see definitions above).

The pause statement is interesting, since it is the only one that creates actions for the control flow. The surface part of the compilation detects when a label is activated: each time we hit a ℓ : `pause` statement, we take the activation condition computed so far and take this for the creation of a new guarded action setting ℓ. The label ℓ can be also activated later in the depth. This is the case if the control is currently at this label and the outer context requests the suspension. The

fun $\mathsf{ControlFlow}(\mathrm{st}, S)$
$(I, \mathscr{C}^{\mathsf{s}}) = \mathsf{CtrlSurface}(\mathrm{st}, S)$;
$(A, T, \mathscr{C}^{\mathsf{d}}) = \mathsf{CtrlDepth}(\mathsf{false}, \mathsf{false}, \mathrm{S})$;
return$(\mathscr{C}^{\mathsf{s}} \cup \mathscr{C}^{\mathsf{d}})$

fun $\mathsf{CtrlSurface}(\mathrm{st}, S)$
switch(S)
case $[\ell : \texttt{pause}]$
return$(\mathsf{false}, \{\mathrm{st} \Rightarrow \mathsf{next}(\ell) = \mathsf{true}\})$

case $[\texttt{if}(\sigma)\, S_1 \texttt{else}\, S_2]$
$(I_1, \mathscr{C}_1^{\mathsf{s}}) = \mathsf{CtrlSurface}(\mathrm{st} \wedge \sigma, S_1)$;
$(I_2, \mathscr{C}_2^{\mathsf{s}}) = \mathsf{CtrlSurface}(\mathrm{st} \wedge \neg\sigma, S_2)$;
return$(I_1 \wedge \sigma \vee I_2 \wedge \neg\sigma, \mathscr{C}_1^{\mathsf{s}} \cup \mathscr{C}_2^{\mathsf{s}})$

case $[S_1; S_2]$
$(I_1, \mathscr{C}_1^{\mathsf{s}}) = \mathsf{CtrlSurface}(\mathrm{st}, S_1)$;
$(I_2, \mathscr{C}_2^{\mathsf{s}}) = \mathsf{CtrlSurface}(\mathrm{st} \wedge I1, S_2)$;
return$(I_1 \wedge I_2, \mathscr{C}_1^{\mathsf{s}} \cup \mathscr{C}_2^{\mathsf{s}})$

case $[\texttt{abort}\, S_1\, \texttt{when}(\sigma)]$
return$\mathsf{CtrlSurface}(\mathrm{st}, S_1)$

$\vdots$

fun $\mathsf{CtrlDepth}(\mathrm{ab}, \mathrm{sp}, \mathrm{S})$
switch(S)
case $[\ell : \texttt{pause}]$
return$(\ell, \ell, \{\ell \wedge \mathrm{sp} \Rightarrow \mathsf{next}(\ell) = \mathsf{true}\})$

case $[\texttt{if}(\sigma)\, S_1 \texttt{else}\, S_2]$
$(A_1, T_1, \mathscr{C}_1^{\mathsf{d}}) = \mathsf{CtrlDepth}(\mathrm{ab}, \mathrm{sp}, \mathrm{S}_1)$;
$(A_2, T_2, \mathscr{C}_2^{\mathsf{d}}) = \mathsf{CtrlDepth}(\mathrm{ab}, \mathrm{sp}, \mathrm{S}_2)$;
return$(A_1 \vee A_2, T_1 \vee T_2, \mathscr{C}_1^{\mathsf{d}} \cup \mathscr{C}_2^{\mathsf{d}})$

case $[S_1; S_2]$
$(A_1, T_1, \mathscr{C}_1^{\mathsf{d}}) = \mathsf{CtrlDepth}(\mathrm{ab}, \mathrm{sp}, \mathrm{S}_1)$;
$\mathrm{st}_2 = T_1 \wedge \neg(sp \vee ab)$;
$(I_2, \mathscr{C}_2^{\mathsf{s}}) = \mathsf{CtrlSurface}(\mathrm{st}_2, S_2)$;
$(A_2, T_2, \mathscr{C}_2^{\mathsf{d}}) = \mathsf{CtrlDepth}(\mathrm{ab}, \mathrm{sp}, \mathrm{S}_2)$;
return$(A_1 \vee A_2, T_1 \wedge I_2 \vee T_2, \mathscr{C}_1^{\mathsf{d}} \cup \mathscr{C}_2^{\mathsf{s}} \cup \mathscr{C}_2^{\mathsf{d}})$

case $[\texttt{abort}\, S_1\, \texttt{when}(\sigma)]$
$(A_1, T_1, \mathscr{C}_1^{\mathsf{d}}) = \mathsf{CtrlDepth}(\mathrm{ab} \vee \sigma, \mathrm{sp}, \mathrm{S}_1)$;
return$(A_1, T_1 \vee A_1 \wedge \sigma, \mathscr{C}_1^{\mathsf{d}})$

$\vdots$

Fig. 2.6 Compiling the control flow (excerpt)

computation of the control flow predicates reveals no surprises: ℓ : `pause` always needs time $\mathsf{inst}() = \mathsf{false}$, it is active if the control flow is currently at label ℓ ($\mathsf{insd}() = \ell$) and it terminates whenever it is active ($\mathsf{term}() = \ell$).

Let us now consider a compound statement like a conditional statement. For the surface, we update the activation condition by adding the guard. Then, the substatements are compiled and the results are merged straightforwardly. The parallel statement (not shown in the figure) is compiled similarly. A bit more interesting is the compilation of the sequence. As already mentioned above, it implements the stepwise traversal of the synchronous program. This is accomplished by the surface calls in the depth. A similar behavior can also be found in the functions for the compilation of the loops. Preemption is also rather simple: since the abortion condition is usually not checked when entering the abort statement, it does not have any influence and can be neglected for the compilation of the surface. In the depth, the abort condition is just appended to the previous one. Finally, suspension is compiled similarly.

The compilation of the control flow works in a linear pass over the syntax tree of the program and generates a set of guarded actions of a linear size with respect to the size of the given program.

2.3.4 Compilation of the Data Flow

Figure 2.7 shows some of the functions which compute the data flow for a given Quartz statement where we make use of a virtual boot label ℓ_0. The surface actions can be executed in the current step, while the actions of the depth are enabled by the resumption of the control flow that already rests somewhere in the statement. For this reason, we do not need a precondition as argument for the depth data flow compilation since the precondition is the active current control flow location itself. The compilation of the surface of a basic statement should be clear: we take the so-far computed precondition st as the guard for the atomic action. The depth variants do not create any actions since statements without control flow locations do not have depth actions.

For the conditional statement, we simply add σ or its negation to the precondition to start the corresponding substatement. As the control flow can rest in one of the branches of an if-statement, it can be resumed from any of these branches. In the depth, we therefore simply take the "union" of the two computations of the depth actions.

```
fun DataFlow(S)
    𝒟^s = DataSurface(ℓ_0, S);
    𝒟^d = DataDepth(S);
    return(𝒟^s ∪ 𝒟^d)

fun DataSurface(st, S)
  switch(S)
    case [x = τ]
        return{st ⇒ x = τ}

    case [next(x) = τ]
        return{st ⇒ next(x) = τ}

    case [if(σ) S1 else S2]
        𝒟^s_1 = DataSurface(st ∧ σ, S1)
        𝒟^s_2 = DataSurface(st ∧ ¬σ, S2)
        return(𝒟^s_1 ∪ 𝒟^s_2)

    case [S1; S2]
        𝒟^s_1 = DataSurface(st, S1)
        𝒟^s_2 = DataSurface(st ∧ inst(S1), S2)
        return(𝒟^s_1 ∪ 𝒟^s_2)

    case [abort S1 when(σ)]
        return DataSurface(st, S1)

    case [weak abort S1 when(σ)]
        return DataSurface(st, S1)
          ⋮

fun DataDepth(S) =
  switch(S)
    case [x = τ]
        return{}

    case [next(x) = τ]
        return{}

    case [if(σ) S1 else S2]
        𝒟^d_1 = DataDepth(S1)
        𝒟^d_2 = DataDepth(S2)
        return(𝒟^d_1 ∪ 𝒟^d_2)

    case [S1; S2]
        𝒟^d_1 = DataDepth(S1)
        𝒟^s_2 = DataSurface(term(S1), S2)
        𝒟^d_2 = DataDepth(S2)
        return(𝒟^d_1 ∪ 𝒟^s_2 ∪ 𝒟^d_2)

    case [abort S1 when(σ)]
        return
          {γ ∧ ¬σ ⇒ 𝒞 | (γ ⇒ 𝒞) ∈ DataDepth(S1)}

    case [weak abort S1 when(σ)]
        return DataDepth(S1)
          ⋮
```

Fig. 2.7 Compiling the data flow (excerpt)

According to the semantics of a sequence, we first execute S_1. If the execution of S_1 is instantaneous, then we also execute S_2 in the same macro step. Hence, the precondition for the surface actions of S_2 is $\varphi \wedge \mathsf{inst}(S_1)$. The preconditions of the substatements of a parallel statement are simply the preconditions of the parallel statement.

In the depth, the control flow can rest in either one of the substatements S_1 or S_2 of a sequence $S_1; S_2$, and hence, we can resume it from either S_1 or S_2. If the control flow is resumed from somewhere inside S_1, and S_1 terminates, then also the surface actions of S_2 are executed in the depth of the sequence. Note that the computation of the depth of a sequence $S_1; S_2$ leads to the computation of the surface actions of S_2 as in the computation of the control flow in the previous section.

As delayed abortions are ignored at starting time of a delayed abort statement, we can ignore them for the computation of the surface actions. Weak preemption statements can be also ignored for the computation of the depth actions, since even if the abortion takes place, all actions remain enabled due to the weak preemption. For the depth of strong abortion statements, we add a conjunct $\neg\sigma$ to the guards of all actions to disable them in case σ holds.

Obviously, the compilation of the control flow (see Fig. 2.6) and the compilation of the data flow (see Fig. 2.7) can be merged into a single set of functions that simultaneously compile both parts of the program. Since the guards of the actions for the data flow refer to the control flow predicates, this approach simplifies the implementation. The result is an algorithm which runs in time $O(|S|^2)$, since $\mathsf{DataSurface}(S,)$ runs in $O(|S|)$ and $\mathsf{DataDepth}(S)$ in $O(|S|^2)$. The reason for the quadratic blow-up is that sequences and loops necessarily have to generate copies of surfaces of their substatements.

2.3.5 Local Variables and Schizophrenia

The characteristic property of local variables is their limited scope. In the context of synchrony, which groups a number of micro steps into an instantaneous macro step, a limited scope which does not match with the macro steps may cause problems. In particular, this is the case if a local declaration is left and reentered within the same macro step, e.g., when a local declaration is nested within loop statements. In such a problematic macro step, the micro steps must then refer to the right incarnation of the local variable since its incarnations in the old and the new scope may have different values in one macro step.

Figure 2.8a shows a simple example. The local variable x, which is declared in the loop body, is referenced at the beginning and at the end of the loop. In the second step of the program, when it resumes from the label ℓ_1, all actions are executed, but they refer to two different incarnations of x: While the assignment to x is made to the old variable in the depth, the if-statement checks the value of the new incarnation which is false.

While software synthesis of sequential programs can solve this problem simply by shadowing the incarnations of the old scope, this is not possible for the

a

```
do {
    bool x;
    if(x) y = 1;
    ℓ1 : pause;
    x = true;
} while(true) ;
```

b

```
do {
    bool x;
    if(x) y = 1;
    if(a) ℓ1 : pause;
    x = true;
    if(¬a) ℓ2 : pause;
} while(true) ;
```

Fig. 2.8 Schizophrenic Quartz programs

synchronous MoC, since each variable has exactly one value per macro step. Therefore, we have to generate a copy of the locally declared variable and map the actions of the program to the corresponding copy in the intermediate code. Furthermore, we have to create additional actions in the intermediate code that link the copies so that the value of the new incarnation at the beginning of the scope is eventually transported to the old one, which is used in the rest of the scope.

However, the problem can be even worse: first, whereas in the previous example each statement always referred to the same incarnation (either the old or the new one), the general case is more complicated as can be seen in Fig. 2.8b. The statements between the two pause statements are sometimes in the context of the old and sometimes in the context of the new incarnation. Therefore, these statements are usually called *schizophrenic* in the synchronous languages community [4]. Second, there can be several reincarnations of a local variable, since the scope can be reentered more than once. In general, the number of loops, which are nested around a local variable declaration determines an upper bound on the number of possible reincarnations.

A challenging Quartz program containing local variables is shown on the left-hand side of Fig. 2.9. The right-hand side of the same figure shows the corresponding control flow graph. The circle nodes of this graph are control flow states that are labeled with those location variables that are currently active (including the start location ℓ_0). Besides these control flow states, there are two other kinds of nodes: boxes contain actions that are executed when an arc toward this node is traversed, while the diamonds represent branches that influence the following computations. The outgoing arcs of such a node correspond to the *then* (solid) and *else* (dashed) branch of the condition. For example, if the program is executed from state ℓ_1 and we have $\neg k \wedge j \wedge \neg i$, then we execute the two action boxes beneath control state ℓ_1 and additionally the one below condition node j.

As can be seen, the condition $k \wedge j \wedge \neg i$ executes all possible action nodes while traversing from control node ℓ_1 to itself. The first action node belongs to the depth of all local declarations, the second one (re)enters the local declaration of c, but remains inside the local declarations of b and a. A new incarnation c_3 is thereby created. The node below condition node k (re)enters the local declarations of b and c, but remains in the one of a. Hence, it creates new incarnations b_2 and c_2 of b

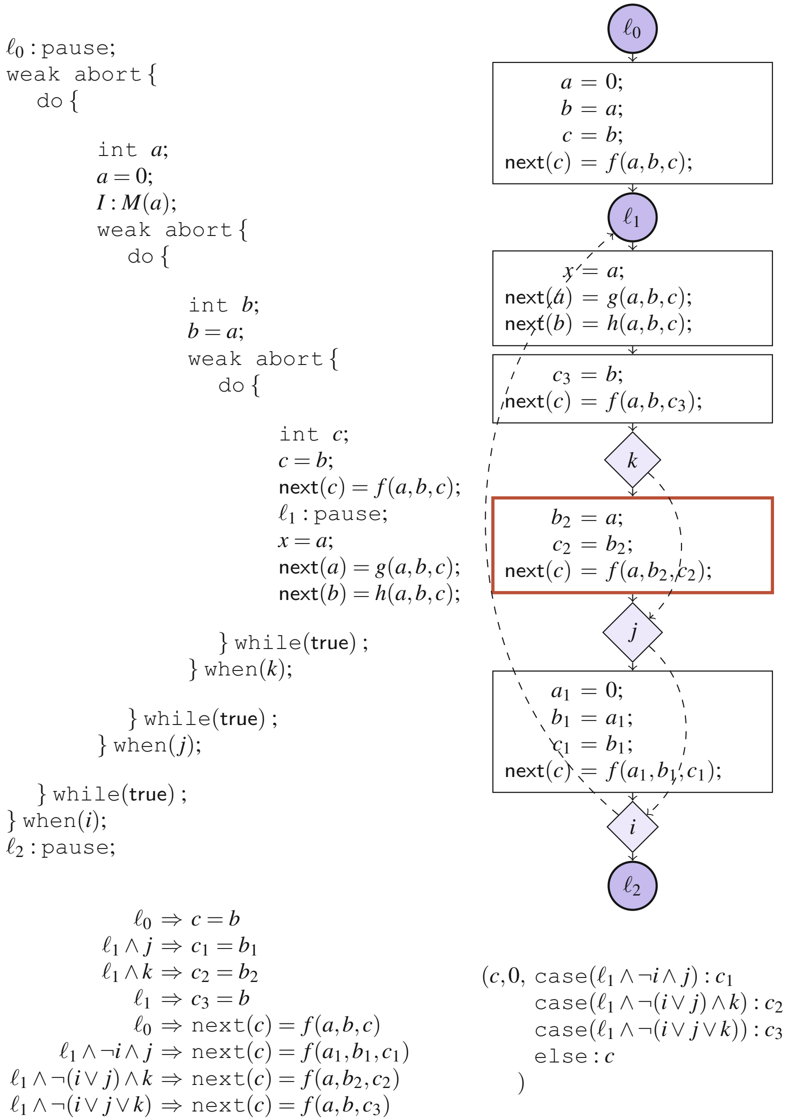

Fig. 2.9 Local declarations with multiple simultaneous reincarnations

and c, respectively. Finally, the remaining node (re)enters all local declarations and therefore generates three incarnations a_1, b_1, and c_1. Note that these four action boxes can be executed at the same point of time, and therefore, the reincarnations a_1, b_1, c_1, b_2, c_2, and c_3 may all exist in one macro step.

Several solutions have been proposed for the solution of schizophrenic statements [4, 35, 41, 49, 54]. Our Quartz compiler carefully distinguishes between the different surfaces of the same schizophrenic local variable and creates fresh incarnations for each one. Technically, this is handled by adding a counter to the compilation functions (Figs. 2.6 and 2.7), which counts how often such a surface has already been entered in the same reaction. Giving all technical details is beyond the scope of this chapter. The interested reader is referred to [11, 47], where the complete solution is described.

2.4 Semantic Analysis

The synchronous MoC abstracts from the execution order within a reaction and postulates that all actions are executed in zero time – at least in the programmer's view. In practice, this means that all actions must be executed according to their data dependencies in order to keep the illusion of zero time execution for the programmer. However, it could happen that there is no such execution order since there are cyclic data dependencies. These so-called causality cycles occur if the input for an action is instantaneously modified by the output of this action or others that were triggered by its output. From the practical side, cyclic programs are rather rare, but they can appear and must therefore be handled by compilers. As a consequence, *causality analysis* [4, 7, 13, 24, 34, 45, 48, 52] must be performed to detect critical cyclic programs.

In general, cyclic programs might have no behavior (loss of reactivity), more than one behavior (loss of determinism), or a unique behavior. However, having a unique behavior is not sufficient for causality, since there are programs whose unique behavior can only be found by guessing. For this reason, causality analysis checks whether a program has a unique behavior that can furthermore be constructively determined by the operational semantics of the program. To this end, the causality analysis starts with known input variables and yet unknown local/output variables. Then, it determines the micro steps of a macro step that can be executed with this incomplete knowledge of the current variables' values. The execution of these micro steps may reveal the values of some further local/output variables of this macro step so that further micro steps can be executed after this round. If all variables became finally known by this fixpoint iteration, the program is a constructive one with a unique behavior.

While causality analysis may appear to be a special problem for synchronous languages, a closer look at the problem reveals that there are many equivalent problems: (1) Shiple [51–53] proved the equivalence to Brzozowski and Seger's timing analysis in the up-bounded inertial delay model [13]. This means that a circuit derived from a cyclic equation system will finally stabilize for arbitrary gate delays iff the equation system is causally correct. (2) Berry pointed out that causality analysis is equivalent to theorem proving in intuitionistic (constructive) propositional logic and introduced the notion of constructive circuits [5]. (3) The problem is also equivalent to type-checking in functional programs due to the

Curry-Howard isomorphism [26]. (4) Finally, Edwards reformulates the problem as the existence of dynamic schedules for the execution of mutually dependent threads of micro steps [20]. Hence, causality analysis is a fundamental analysis that has already found many applications in computer science.

We will not discuss the details in this chapter. The interested reader is referred to [45], where all details about the causality analysis performed in the context of Quartz can be found.

2.5 Synthesis

Before presenting the synthesis procedures, we first recall our overall design flow that determines the context of the compilation procedure. As we target the design of embedded systems, where hardware-software partitioning and target platforms are design decisions that are frequently changed, persistent intermediate results in a well-defined and robust format are welcome. In our Averest system, we basically split the design flow into two steps, which are bridged by the AIF. This intermediate format captures the system behavior in terms of synchronous guarded actions. Hence, complex control flow statements need no longer be considered. We refer to *compilation* as the translation of source code into AIF, while *synthesis* means the translation from AIF to the final target code, which may be based on a different MoC.

Figure 2.10 shows two approaches of generating target code from a set of Quartz modules. *Modular compilation*, which is shown on the left-hand side, translates each Quartz module to a corresponding AIF module. Then, these modules are linked on the intermediate level before the whole system is synthesized to target code. *Modular synthesis*, which is shown on the right-hand side, translates each Quartz module to a corresponding AIF module, which is subsequently synthesized to a target code module. Linking is then deferred to the target level. While modular synthesis simplifies the compilation (since all translation processes have to consider only a module of the system), it puts the burden on the run-time platform or the linker which has to organize the interaction of the target modules correctly.

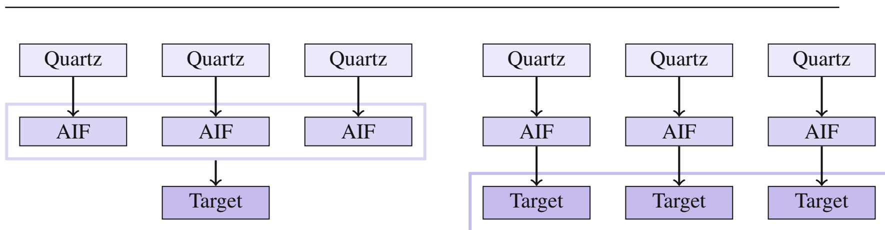

Fig. 2.10 Modular compilation and modular synthesis of Quartz programs

From our intermediate representation of guarded actions, many synthesis targets can be thought of. In the following, we sketch the translation to different targets; first to symbolic transition systems, which are suitable for formal verification of program properties by symbolic model checking, second to digital hardware circuits for hardware synthesis, third the translation to SystemC code, which can be used for an integrated simulation of the system, and finally an automaton-based sequential software synthesis.

2.5.1 Symbolic Model Checking

For symbolic model checking, the system generally needs to be represented by a transition system. This basically consists of a triple $(\mathscr{S}, \mathscr{I}, \mathscr{T})$ with a set of states $\mathscr{S}$, initial states $\mathscr{I} \subseteq \mathscr{S}$, and a transition relation $\mathscr{T} \subseteq \mathscr{S} \times \mathscr{S}$. Each state s is a mapping from variables to values, i.e., s assigns to each variable a value of its domain. As we aim for a symbolic description, we describe the initial states and the transition relation by propositional formulas $\Phi_{\mathscr{I}}$ and $\Phi_{\mathscr{T}}$, which are their characteristic functions.

For the presentation of the translation, assume that our intermediate representation contains immediate and delayed actions for each variable x of the following form:

$$
\begin{array}{ll}
(\gamma_1, \texttt{x} = \tau_1), & \ldots, (\gamma_p, \texttt{x} = \tau_p) \\
(\chi_1, \texttt{next(x)} = \pi_1), & \ldots, (\chi_q, \texttt{next(x)} = \pi_q)
\end{array}
$$

Figure 2.11 sketches the translation of the immediate and delayed actions writing variable x to clauses used for the description of a symbolic transition system.

The construction of a transition system is quite straightforward: The initial value of a variable x can only be determined by its immediate actions. Hence, if one of the guards γ_i of the immediate actions holds, the corresponding immediate assignment defines the value of x. If none of the guards γ_i should hold, the initial value of x is determined by its default value (which is determined by the semantics, e.g., false for Boolean variables and 0 for numeric ones).

The transition relation determines which states can be connected by a transition, i.e., which values the variables may have at the next point of time given values at the current point of time. To this end, the transition relation sets up constraints for the values at the current and the next point of time: First, also the immediate assignments have to be respected for the current point of time, i.e., whenever a guard γ_i of the immediate actions holds, the corresponding immediate assignment defines the current value of x. If one of the guards χ_i of the delayed assignments hold at the current point of time, the next value of x is determined by the corresponding delayed assignment. Finally, if the next value of x is not determined by an action, i.e., none of the guards γ_i of the immediate assignments hold at the next point of time and neither holds one of the guards χ_i at the current point of time, then the next value of x is determined by the reaction to absence.

$$\mathsf{Init}_x :\equiv \left(\begin{array}{l} \bigwedge_{j=1}^{p} (\gamma_j \to x = \tau_j) \wedge \\ \\ \left(\bigwedge_{j=1}^{p} \neg\gamma_j \right) \to x = \mathsf{Default}(x) \end{array} \right)$$

$$\mathsf{Trans}_x :\equiv \left(\begin{array}{l} \bigwedge_{j=1}^{p} (\gamma_j \to x = \tau_j) \wedge \\ \\ \bigwedge_{j=1}^{q} (\chi_j \to \mathsf{next}(x) = \pi_j) \wedge \\ \\ \mathsf{next}(\bigwedge_{j=1}^{p} \neg\gamma_j) \wedge \left(\bigwedge_{j=1}^{q} \neg\chi_j \right) \to \mathsf{next}(x) = \mathsf{Abs}(x) \end{array} \right)$$

$$\mathsf{Abs}(x) := \begin{cases} \mathsf{Default}(x) & : \text{if } x \text{ is an event variable} \\ x & : \text{if } x \text{ is a memorized variable} \end{cases}$$

Fig. 2.11 Transition relation for x

The definitions given in Fig. 2.11 can be literally used to define input files for symbolic model checkers. Causality problems do not bother in this translation, and also write conflicts will show up as deadend states in the transition diagram and can be checked this way by symbolic model checking.

2.5.2 Circuit Synthesis

The transition relation shown in Fig. 2.11 of the previous section can be modified so that both the initial condition and the transition relation become equation systems provided that there are no write conflicts between the actions of any variable x. To explain the construction of the equations, we consider again any variable x with the following guarded actions:

$$\begin{array}{l} (\gamma_1, \mathtt{x} = \tau_1), \qquad\qquad \ldots, (\gamma_p, \mathtt{x} = \tau_p) \\ (\chi_1, \mathtt{next(x)} = \pi_1), \ldots, (\chi_q, \mathtt{next(x)} = \pi_q) \end{array}$$

The equations for x are shown in Fig. 2.12 where an additional carrier variable x' is used. This carrier variable x' has initially the default value of x, and its value at any next point of time is determined by the delayed assignments of x. If none of the delayed assignments is enabled, the reaction to absence is applied, i.e., $\mathsf{next}(x') = \mathsf{Abs}(x)$ will hold.

For the variable x itself, we introduce only an immediate equation where the current value of x is defined by the immediate assignments to it, and if none of them is enabled, we use the current value of the carrier variable to define x.

One can prove that x has this way the correct value for any point of time by induction over time: At the initial point of time, one of the immediate assignments will determine the initial value of x if one is enabled, which is equivalent to the initial condition of Fig. 2.11. If none of the γ_i holds, we have $x := x' := \mathsf{Default}(x)$ which is also the case for the initial condition of Fig. 2.11.

To determine the value of x at any later point of time, note first that again the immediate assignments can determine its value. If none of them is enabled, we have again $x := x'$. If we consider this equation from the previous point of time, it means $\mathsf{next}(x) = \mathsf{next}(x')$ so that we can see that x may be determined by delayed assignments that were enabled in the previous point of time. Finally, if none of these were enabled either, then the reaction to absence takes place which had already determined the value of x' so that $x := x'$ will define the value of x correctly also in this case.

Note that the equation system as given in Fig. 2.12 has only immediate equations for the output and local variables, while carrier variables are used to capture the delayed assignments. Since the carrier variables are not observable outside the module, they define the internal state together with the local variables of the program and the control flow labels of the pause statements. Note that these equation systems have exactly the form we assumed in the causality analysis described in Sect. 2.4.

Finally, we note that the actual synthesis of the equations is much more difficult due to the reincarnation of local variables. Since the reincarnations do only occur in the surface statements, and since surface statements can be executed in zero time, the behavior of the reincarnated variables can be described by simple immediate equations without carrier variables. However, the reaction to absence is more complicated for the local variable that remains in the depth of a local declaration statement: Here we have to determine which of the different surfaces has been the latest one executed that carries then its value to the depth. Furthermore, the delayed assignments to local variables in those reincarnated surfaces that are followed by further reincarnations have to be disabled. The overall procedure is quite difficult and is described in full detail in [43, 46].

$$\mathsf{init}\,(x') = \mathsf{Default}(x)$$

$$\mathsf{next}(x') = \left(\begin{array}{l}\textbf{case}\\ \qquad \chi_1 : \pi_1;\\ \qquad \quad \vdots\\ \qquad \chi_q : \pi_q;\\ \textbf{else } \mathsf{Abs}(x)\end{array}\right) \qquad x = \left(\begin{array}{l}\textbf{case}\\ \qquad \gamma_1 : \tau_1;\\ \qquad \quad \vdots\\ \qquad \gamma_p : \tau_p;\\ \textbf{else } x'\end{array}\right)$$

Fig. 2.12 Equation system for x

2.5.3 SystemC Simulation

The simulation semantics of SystemC is based on the discrete-event model of computation [14], where reactions of the system are triggered by events. All threads that are sensitive to a specified set of events are activated and produce new events during their execution. Updates of variables are not immediately visible, but become visible in the next delta cycle.

We start the translation by the definition of a global clock that ticks in each instant and drives all the computation. Thus, we require that the processed model is *endochronous* [22, 23], i.e., there is a signal which is present in all instants of the behavior and from which all other signals can be determined. In SystemC, this clock is implemented by a single `sc_clock` at the uppermost level, and all other components are connected to this clock. Hence, the translations of the macro steps of the synchronous program in SystemC are triggered by this clock, while the micro steps are triggered by signal changes in the delta cycles. For this reason, input and output variables of the synchronous program are mapped to input signals (`sc_in`) and output signals (`sc_out`) of SystemC of the corresponding type.

Additionally, we declare signals for all other clocks of the system. They are inputs since the clock constraints (as given by **`assume`**) do not give an operational description of the clocks, but can be only checked in the system. The clock calculus for Signal [1, 22, 23] or scheduler creation for CAOS [12] aim at creating exactly these schedulers which give an operational description of the clocks. Although not covered in the following, their result can be linked to the system description so that clocks are driven by the system itself.

The translation of the synchronous guarded actions to SystemC processes is however not as simple as one might expect. The basic idea is to map guarded actions to methods which are sensitive to the read variables so that the guarded action is re-evaluated each time one of the variables it depends on changes. For a *constructive* model, it is guaranteed that the simulation does not hang up in delta cycles.

The translation to SystemC must tackle the following two problems: (1) As SystemC does not allow a signal to have multiple drivers, all immediate and delayed actions must be grouped by their target variables, or equivalently, we can produce the equations as shown in the previous section. (2) The SystemC simulation semantics can lead to spurious updates of variables (in the AIF context), since threads are always triggered if some variables in the sensitivity list have been updated – even if they are changed once more in later delta cycles. As actions might be spuriously activated, it must be ensured that at least one action is activated in each instant, which sets the final value. Both problems are handled in a similar way as the translation to the transition system presented in the previous section: we create an additional variable `_carrier_x` for each variable x to record values from their delayed assignments and group all actions in the same way as for the transition system.

With these considerations, the translation of the immediate guarded actions $\langle \gamma \Rightarrow x = \tau_i \rangle$ is straightforward: We translate each group of actions into an asynchronous thread in SystemC, which is sensitive to all signals read by these actions (variables appearing in the guards γ_i or in the right-hand sides τ_i). Thereby, all actions are

```
void Module                              void Module
  ::compute_x()                            ::compute_delayed_x()
{                                        {
  while(true) {                            while(true) {
    if(_clk_x.read() && γ1)                  if(ξ1)
      x.write(τ1);                             _carrier_x.write(π1);
      ...                                      ...
    else if(_clk_x.read() && γn)             else if(ξn)
      x.write(τn);                             _carrier_x.write(πn);
    else if(_clk_x.read())                   else
      x.write(_carrier_x.read());              _carrier_x.write(x.read());
    wait();                                  wait();
  }                                        }
}                                        }
```

Fig. 2.13 SystemC: Translation of immediate and delayed actions

implemented by an `if`-block except for the last one, which handles the case that no action fires. Since the immediate actions should become immediately visible, the new value can be immediately written to the variable with the help of a call to `x.write(...)`. Analogously, the evaluations of the guard γ_i and the right-hand side of the assignment τ_i make use of the `read` methods of the other signals. The left-hand side of Fig. 2.13 shows the general structure of such a thread.

Delayed actions $\langle \gamma \Rightarrow \texttt{next}(x) = \pi_j \rangle$ are handled differently: While the right-hand side is immediately evaluated, the assignment should only be visible in the following macro step and not yet in the current one. Hence, they do not take part in the fixpoint iteration. Therefore, we write their result to `_carrier_x` in a *clocked thread*, which is triggered by the master trigger Thereby, signals changed by the delayed actions do not affect the current fixpoint iteration and vice versa.

Figure 2.14 gives the SystemC code for the part that simulates the variables x. Similar to the Symbolic Model Verifier (SMV) translation, we abbreviate guards for reuse in different SystemC processes. Then, the translation of the immediate actions writing x is straightforward; they correspond to the first two cases in method `OuterQuartz::compute_x()`. The last case is responsible for setting x to its previous value if neither of the two immediate actions fires. As already stated above, we need to do this explicitly. To this end, the previous value of variable x is always stored in a separate carrier variable. For variables, which are only set by delayed actions, we can simplify the general scheme of Fig. 2.13. In this case, we can combine the two threads as Fig. 2.14 illustrates: we only need a single variable, which is set by this thread.

2.5.4 Automaton-Based Sequential Software Synthesis

In order to generate fast sequential code from synchronous programs, the Extended Finite-State Machine (EFSM) representation of the program is an ideal starting

```
void OuterQuartz
  ::compute_x()
{
  while(true) {
    if(_grd18)
      x.write(y.read());
    else if(_grd19)
      x.write(2*y.read());
    else if(_clk_x.read())
      x.write(_carrier_x.read());
    wait();
  }
}
```

```
void OuterQuartz
  ::compute_delayed_x()
{
  while(true) {
    _carrier_x.write(x.read());
    wait();
  }
}

void OuterQuartz
  ::compute_delayed_ell1()
{
  while(true) {
    if(_grd18)
      ell1.write(true);
    else
      ell1.write(false);
    wait();
  }
}
```

Fig. 2.14 SystemC code

point. EFSMs explicitly represent the state transition system of the control flow: each state s represents a subset $\mathsf{Labels}(s) \subseteq \mathscr{L}$ of the control flow labels, and edges between states are labeled with conditions that must be fulfilled to reach the target state from the source state. EFSMs are therefore a representation where the control flow of a program state is explicitly represented, while the data flow is still represented symbolically (while synchronous guarded actions represent control flow and data flow symbolically).

The guards of the guarded actions of the control flow are therefore translated to transition conditions of the EFSM's state transitions. The guarded actions of the data flow are first copied to each state of the EFSM and are then partially evaluated according to the values of the control flow labels in that EFSM state. Hence, in each macro step, the generated code will only consider a subset $D(s)$ of the guarded actions, which generally speeds up the execution (since many of them are only active in a small number of states).

Definition 1 (Extended Finite State Machine). An *Extended Finite-State Machine (EFSM)* is a tuple (S, s_0, T, D), where S is a set of states, $s_0 \in S$ is the initial state, and $T \subseteq (S * C * S)$ is a finite transition relation where C is the set of transition conditions. D is a mapping $S \to \mathscr{D}$, which assigns each state $s \in S$ a set of data flow guarded actions $D(s) \subseteq \mathscr{D}$ which are executed in state s.

To see an example of an EFSM, consider first the Quartz program shown in Fig. 2.15. It computes the integer approximation of the euclidean length of a N-dimensional vector v and does only make use of addition, subtraction, and

comparison. The correctness is not difficult to see by the given invariants in the comments:

```
next(x[i]) * next(y[i]) + next(p[i])
     = x[i] * (y[i] - 1) + (p[i] + x[i])
     = x[i] * y[i] + p[i]

next(y[0])
     = y[0] + x[0] + x[0] + 3
     = (x[0]+1)^2 + 2*x[0] + 3
     = x[0]^2 + 2*x[0] + 1 + 2*x[0] + 3
     = x[0]^2 + 4*x[0] + 4
     = (x[0]+2)^2
     = (next(x[0])+1)^2
```

The corresponding EFSM of the program shown in Fig. 2.15 is shown in Fig. 2.16 with seven states. Each state is given an index and lists the guarded actions that can

```
macro N = 2;

module VectorLength([N]int v, int !len, event !rdy) {
     [N]int p, x, y;
     // compute the squares of each v[i] in p[i] in parallel
     for(i=0..N-1) do || {
          x[i] = v[i];
          y[i] = v[i];
          p[i] = 0;
          while(y[i]>0) {
               // invariant: v[i]*v[i] = x[i]*y[i]+p[i]
               next(p[i]) = p[i] + x[i];
               next(y[i]) = y[i] - 1;
               w0: pause;
          }
     }
     // add all p[i] in p[0]
     next(p[0]) = sum(i=0..N-1) p[i];
     w1: pause;
     // now compute the square root of p[0]
     x[0] = 0;
     y[0] = 1;
     while(y[0]<=p[0]) {
          // invariant: y[0] = (x[0]+1)^2
          next(x[0]) = x[0] + 1;
          next(y[0]) = y[0] + x[0] + x[0] + 3;
          w2: pause;
     }
     // return the length of the vector v
     emit(rdy);
     len = x[0];
}
```

Fig. 2.15 The VectorLength Quartz program

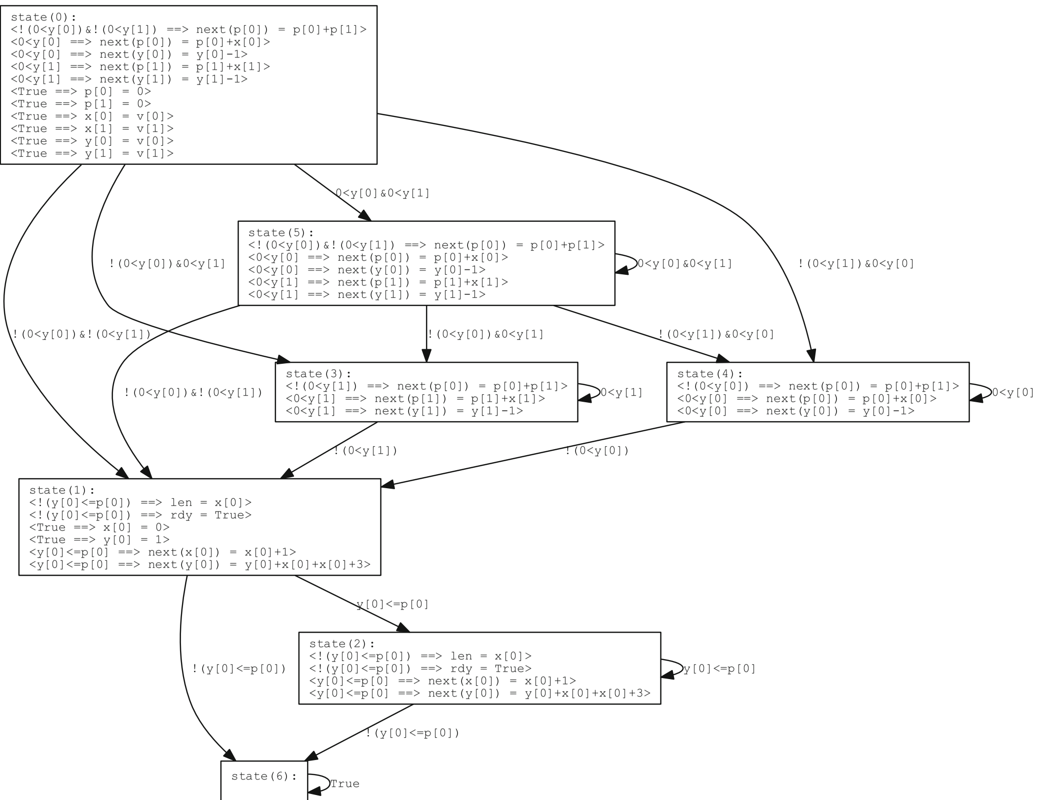

Fig. 2.16 The EFSM of `VectorLength` Quartz program for *two*-dimensional vectors

be enabled in that control flow state. State 0 contains thereby the surface of the program where the initialization of variables `x,y,p` of the for-loop and possibly the assignments in its while-loop are executed. In state 5, both while-loops for computing the squares of `v[0]` and `v[1]` are active, while in state 3 and state 4, only the one for computing the squares of `v[1]` and `v[0]` continues with the execution. Note that these states may also execute the summation **`next`**`(p[0])= p[0]+p[1]` in case the loop terminates. State 1 corresponds with label `w1` and executes the code between `w1` and `w2`, and finally state 2 executes the code from `w2` back to `w2` or leaving the loop. State 6 is a final sink state that is always added for technical reasons.

A naive way to generate the EFSM states is to take all possible 2^n states and compute the transitions from each one. However, as many states are generally not reachable, that algorithm would always need exponential time, even for programs that lead to compact EFSMs. Therefore, a better way to compute the EFSM is an abstract simulation of the program according to the operational semantics of the Quartz language.

As the control flow is explicitly enumerated, EFSMs may suffer from state-space explosion since n control flow locations may result in 2^n EFSM states. It is not only the amount of (control flow) states that poses problems, but the guarded actions for the data flow must be also replicated. However, for many practical examples, the EFSM size is still manageable, and due to the performed pre-computation, it can be optimized in many ways and can produce the fastest target code at the end. It is straightforward to generate a sequential program from an EFSM: For example, we can first define for every state a sequential code starting with a unique label and ending with a goto statement to the next code fragment of the corresponding target state.

It is important to see the difference between an EFSM and control flow graphs used in classic compiler design. While "states" of classic control/data flow graphs consist of assignments that are sequentially executed, states of the EFSM contain still guarded actions that are concurrently executed within one macro step. Moreover, transitions in the EFSM terminate a macro step of the synchronous model, so that new values of the input variables are read on the transition. Due to these differences, many transformations made in classic code optimization cannot directly be applied on EFSMs for code generation of synchronous programs.

2.6 Conclusions and Future Extensions

The synchronous model of computation can perfectly model reactive systems since its programming paradigm directly reflects the execution steps of these systems: Within a reaction step, inputs are read, and outputs are immediately computed as the reaction to these inputs. We derived the language Quartz from the classic Esterel language and modified its syntax and semantics to allow a more convenient description of hardware circuits. We also developed a formally verified compilation to synchronous guarded actions that are used as internal representation in our design

framework Averest. Using the guarded actions, various analyses are performed, in particular, the causality analysis, so that robust and deterministic system models are guaranteed. For causally correct systems, we can then generate both hardware circuits and programs, where the latter can now also be done with multiple threads. Future extensions of the language cover clocked signals that can be absent at some points of time which removes the need to synchronize generated software threads between the macro steps by completely desynchronizing the threads. Moreover, the current version of Quartz already supports hybrid systems so that one can also model a discrete system in its physical environment, e.g., for simulation or formally proving important safety properties.

References

1. Benveniste A, Bournai P, Gautier T, Le Guernic P (1985) SIGNAL: a data flow oriented language for signal processing. Research report 378, Institut National de Recherche en Informatique et en Automatique (INRIA), Rennes
2. Benveniste A, Caspi P, Edwards S, Halbwachs N, Le Guernic P, de Simone R (2003) The synchronous languages twelve years later. Proc IEEE 91(1):64–83
3. Berry G (1992) A hardware implementation of pure Esterel. Sadhana 17(1):95–130
4. Berry G (1999) The constructive semantics of pure Esterel. http://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf
5. Berry G (2000) The Esterel v5 language primer. ftp://ftp.inrialpes.fr/pub/synalp/reports/esterel-primer.pdf.gz
6. Berry G, Gonthier G (1992) The Esterel synchronous programming language: design, semantics, implementation. Sci Comput Program 19(2):87–152
7. Boussinot F (1998) SugarCubes implementation of causality. Research report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis
8. Brandt J (2013) Synchronous models for embedded software. Master's thesis, Department of Computer Science, University of Kaiserslautern. Habilitation
9. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin JP (2012) Representation of synchronous, asynchronous, and polychronous components by clocked guarded actions. Des Autom Embed Syst (DAEM). doi:10.1007/s10617-012-9087-9
10. Brandt J, Gemünde M, Schneider K, Shukla S, Talpin JP (2013) Embedding polychrony into synchrony. IEEE Trans Softw Eng (TSE) 39(7):917–929
11. Brandt J, Schneider K (2011) Separate translation of synchronous programs to guarded actions. Internal report 382/11, Department of Computer Science, University of Kaiserslautern, Kaiserslautern
12. Brandt J, Schneider K, Shukla S (2010) Translating concurrent action oriented specifications to synchronous guarded actions. In: Lee J, Childers B (eds) Languages, compilers, and tools for embedded systems (LCTES). ACM, Stockholm, pp 47–56
13. Brzozowski J, Seger CJ (1995) Asynchronous circuits. Springer, New York/Berlin
14. Cassandras C, Lafortune S (2008) Introduction to discrete event systems, 2nd edn. Springer, New York
15. Chandy K, Misra J (1989) Parallel program design. Addison-Wesley, Austin
16. Closse E, Poize M, Pulou J, Sifakis J, Venter P, Weil D, Yovine S (2001) TAXYS: a tool for the development and verification of real-time embedded systems. In: Berry G, Comon H, Finkel A (eds) Computer aided verification (CAV). LNCS, vol 2102. Springer, Paris, pp 391–395
17. Closse E, Poize M, Pulou J, Venier P, Weil D (2002) SAXO-RT: interpreting Esterel semantics on a sequential execution structure. Electron Notes Theor Comput Sci (ENTCS) 65(5):80–94. Workshop on synchronous languages, applications, and programming (SLAP)

18. Dill D (1996) The Murphi verification system. In: Alur R, Henzinger T (eds) Computer aided verification (CAV). LNCS, vol 1102. Springer, New Brunswick, pp 390–393
19. Edwards S (2002) An Esterel compiler for large control-dominated systems. IEEE Trans Comput Aided Des Integr Circuits Syst (T-CAD) 21(2):169–183
20. Edwards S (2003) Making cyclic circuits acyclic. In: Design automation conference (DAC). ACM, Anaheim, pp 159–162
21. Edwards S, Kapadia V, Halas M (2004) Compiling Esterel into static discrete-event code. In: Synchronous languages, applications, and programming (SLAP), Barcelona
22. Gamatie A (2010) Designing embedded systems with the SIGNAL programming language. Springer, New York
23. Gamatié A, Gautier T, Le Guernic P, Talpin J (2007) Polychronous design of embedded real-time applications. ACM Trans Softw Eng Methodol (TOSEM) 16(2), Article 9. http://dl.acm.org/citation.cfm?id=1217298
24. Halbwachs N, Maraninchi F (1995) On the symbolic analysis of combinational loops in circuits and synchronous programs. In: Euromicro conference. IEEE Computer Society, Como
25. Harel D, Naamad A (1996) The STATEMATE semantics of statecharts. ACM Trans Softw Eng Methodol (TOSEM) 5(4):293–333
26. Howard W (1980) The formulas-as-types notion of construction. In: Seldin J, Hindley J (eds) To H.B. Curry: essays on combinatory logic, lambda-calculus and formalism. Academic, London/New York, pp 479–490
27. Järvinen H, Kurki-Suonio R (1990) The DisCo language and temporal logic of actions. Technical report 11, Tampere University of Technology, Software Systems Laboratory
28. Ju L, Huynh B, Chakraborty S, Roychoudhury A (2009) Context-sensitive timing analysis of Esterel programs. In: Design automation conference (DAC). ACM, San Francisco, pp 870–873
29. Ju L, Khoa Huynh, B., Roychoudhury A, Chakraborty S (2010) Timing analysis of Esterel programs on general purpose multiprocessors. In: Sapatnekar S (ed) Design automation conference (DAC). ACM, Anaheim, pp 48–51
30. Lamport L (1991) The temporal logic of actions. Technical report 79, Digital Equipment Cooperation
31. Li YT, Malik S (1999) Performance analysis of real-time embedded software. Kluwer, Boston/Dordrecht
32. Logothetis G, Schneider K (2003) Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In: Design, automation and test in Europe (DATE). IEEE Computer Society, Munich, pp 10196–10203
33. Malik S (1993) Analysis of cyclic combinational circuits. In: International conference on computer-aided design (ICCAD). IEEE Computer Society, Santa Clara, pp 618–625.
34. Malik S (1994) Analysis of cycle combinational circuits. IEEE Trans Comput Aided Des Integr Circuits Syst (T-CAD) 13(7):950–956
35. Poigné A, Holenderski L (1995) Boolean automata for implementing pure Esterel. Arbeitspapiere 964, GMD, Sankt Augustin
36. Potop-Butucaru D, de Simone R (2003) Optimizations for faster execution of Esterel programs. In: Formal methods and models for codesign (MEMOCODE). IEEE Computer Society, Mont Saint-Michel, pp 227–236
37. Potop-Butucaru D, Edwards S, Berry G (2007) Compiling Esterel. Springer, Boston
38. Rocheteau F, Halbwachs N (1992) Implementing reactive programs on circuits: a hardware implementation of LUSTRE. In: de Bakker J, Huizing C, de Roever WP, Rozenberg G (eds) Real-time: theory in practice. LNCS, vol 600. Springer, Mook, pp 195–208
39. Rocheteau F, Halbwachs N (1992) Pollux: a Lustre-based hardware design environment. In: Quinton P, Robert Y (eds) Algorithms and parallel VLSI architectures II. Elsevier, Gers, pp 335–346
40. Schneider K (2000) A verified hardware synthesis for Esterel. In: Rammig F (ed) Distributed and parallel embedded systems (DIPES). Kluwer, Schloß Ehringerfeld, pp 205–214

41. Schneider K (2001) Embedding imperative synchronous languages in interactive theorem provers. In: Application of concurrency to system design (ACSD). IEEE Computer Society, Newcastle Upon Tyne, pp 143–154
42. Schneider K (2002) Proving the equivalence of microstep and macrostep semantics. In: Carreño V, Muñoz C, Tahar S (eds) Theorem proving in higher order logics (TPHOL). LNCS, vol 2410. Springer, Hampton, pp 314–331
43. Schneider K (2009) The synchronous programming language Quartz. Internal report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern
44. Schneider K, Brandt J (2008) Performing causality analysis by bounded model checking. In: Application of concurrency to system design (ACSD). IEEE Computer Society, Xi'an, pp 78–87
45. Schneider K, Brandt J, Schüle T (2004) Causality analysis of synchronous programs with delayed actions. In: Compilers, architecture, and synthesis for embedded systems (CASES). ACM, Washington, pp 179–189
46. Schneider K, Brandt J, Schüle T (2004) A verified compiler for synchronous programs with local declarations (proceedings version). In: Synchronous languages, applications, and programming (SLAP), Barcelona
47. Schneider K, Brandt J, Schüle T (2006) A verified compiler for synchronous programs with local declarations. Electron Notes Theor Comput Sci (ENTCS) 153(4):71–97
48. Schneider K, Brandt J, Schüle T, Türk T (2005) maximal causality analysis. In: Desel J, Watanabe Y (eds) Application of concurrency to system design (ACSD). IEEE Computer Society, Saint-Malo, pp 106–115
49. Schneider K, Wenz M (2001) A new method for compiling schizophrenic synchronous programs. In: Compilers, architecture, and synthesis for embedded systems (CASES). ACM, Atlanta, pp 49–58
50. Schüle T, Schneider K (2004) Abstraction of assembler programs for symbolic worst case execution time analysis. In: Malik S, Fix L, Kahng A (eds) Design automation conference (DAC). ACM, San Diego, pp 107–112
51. Shiple T (1996) Formal analysis of synchronous circuits. PhD thesis, University of California, Berkeley
52. Shiple T, Berry G, Touati H (1996) Constructive analysis of cyclic circuits. In: European design automation conference (EDAC). IEEE Computer Society, Paris, pp 328–333
53. Shiple T, Singhal V, Brayton R, Sangiovanni-Vincentelli A (1996) Analysis of combinational cycles in sequential circuits. In: International symposium on circuits and systems (ISCAS), Atlanta, pp 592–595
54. Tardieu O, de Simone R (2004) Curing schizophrenia by program rewriting in Esterel. In: Formal methods and models for codesign (MEMOCODE). IEEE Computer Society, San Diego, pp 39–48