Separate Translation of Synchronous Programs to Guarded Actions

Jens Brandt and Klaus Schneider
Embedded Systems Group
Department of Computer Science
University of Kaiserslautern
http://es.cs.uni-kl.de

Abstract

This report gives an overview of the separate translation of synchronous imperative programs to synchronous guarded actions. In particular, we consider problems to be solved for separate compilation that stem from preemption statements and local variable declarations. We explain how we solved these problems and sketch our solutions implemented in the our Averest framework to implement a compiler that allows a separate compilation of imperative synchronous programs with local variables and unrestricted preemption statements. The focus of the report is the big picture of our entire design flow.

1 Introduction

Imperative synchronous languages [2] like Quartz [38] or Esterel [4] offer many important features to fulfill the requirements imposed by embedded systems. The common paradigm of these languages is *perfect synchrony* [2], which means that most of the statements are executed as *micro steps* in zero time. Consumption of time is explicitly programmed by grouping finitely many micro steps to macro steps. As a consequence, all threads of the program run in lockstep: they execute the micro steps of the current macro step in zero time, and automatically synchronize at the end of the macro step. As all micro steps of a macro step are executed at the same point of time, their ordering within the macro step is irrelevant. Therefore, values of variables are determined with respect to macro steps instead of micro steps.

The introduction of a logical time scale is not only a very convenient programming model, it is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs. Thus, synchronous programs can be directly executed on simple micro-controllers without using complex operating systems. Another advantage is the straightforward translation of synchronous programs to hardware circuits [3, 34, 38]. Furthermore, the concise formal semantics of synchronous languages makes them particularly attractive for reasoning about program properties and equivalences [5, 35, 36, 37, 40]. Finally, they allow developers to determine tight bounds on the reaction time by a simplified worst-case execution time analysis (since loops do not appear in reaction steps) [25, 26, 45, 46]

However, this abstraction is not for free: the compilation of synchronous languages is more difficult than the compilation of traditional sequential languages. The concurrency available in synchronous programs has to be translated such that the resulting code can be executed on a sequential machine. To this end, special problems like *causality problems* [29, 21, 11, 47, 8, 5, 39, 42] or *schizophrenic problems* [31, 5, 44, 48, 40] must be solved by compilers. Although these problems turned out to be quite difficult, good algorithms have been found for their solution.

However, the solutions found so-far for causality and schizophrenia problems cannot be used in a *modular compilation*. Additionally, one has to consider a preemption context for each module to allow later calls in preemption statements. Hence, modular compilation has to re-consider these problems and has to engineer new solutions. Clearly, a modular approach is desired for many obvious reasons: First, expensive optimizations may have already been performed on the intermediate results. Complex programs that include the interaction of concurrent threads with preemption statements as well as instantaneous broadcast communication can already be reduced to simpler statements in a first compilation phase. Moreover, the intermediate code can be distributed in libraries without revealing its proprietary source [10].

This report therefore gives an overview of the compilation of imperative synchronous programs written in the language Quartz to synchronous guarded actions, which are the basis for the Averest Intermediate Format (AIF). In particular, we describe how we achieved a separate compilation that allows unrestricted use of local variables and preemption statements. Previous papers have made these results only accessible to a small group of experts, whereas this report targets a broader audience: its focus is put on the main ideas used to achieve a separate compilation. For further details, we refer to previous publications [39, 41, 40, 38, 9].

The rest of the report is structured as follows: Section 2 describes the synchronous model of computation, which is the basis for both the source (Quartz) and the target (AIF) code of the translation presented here. Section 3 sketches our overall design flow and defines various activities for code generation from synchronous programs. Section 4 and Section 5 describe the synchronous language Quartz and synchronous guarded actions, which are our source and target languages. In Section 6, we show the main principle of the compilation and extend it in Section 7 to local variable declarations, and in Section 8 to the separate compilation of modules. Finally, the report ends with a short summary in Section 9.

2 The Synchronous Model of Computation

The synchronous *model of computation* [20, 2] assumes that the execution of programs consists of a sequence of reactions $\mathcal{R} = \langle R_0, R_1, \ldots \rangle$. In each of these reactions (often called macro steps [22]), the system reads its inputs and computes and writes its outputs. According to the synchronous model of computation (MoC), the actions that take place within a reaction (often called micro steps) are not ordered. Instead, micro steps are assumed to happen simultaneously, i. e. in the same variable environment. Hence, variables are constant during the execution of the micro steps and only change synchronously at macro steps. From the semantical point of view, which postulates that a reaction is atomic, neither communication nor computation take time in this sense. When executed on a sequential machine, this MoC implies that all actions are executed according to their data dependencies.

Synchronous hardware circuits are the most important systems that are based on the synchronous MoC. There, 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. All variables are usually modeled by registers, which do not change within a clock cycle. All these correspondences make the modeling and synthesis of synchronous circuits from synchronous programs very appealing [3, 5, 34, 35, 40].

Causality cycles occur when the condition for executing an action is instantaneously modified by the result of this action, which is comparable to combinational loops in a hardware circuit. 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* [5] must be performed to detect critical cyclic programs. It checks if such cyclic dependencies have unique solutions, which is related to many other areas, e.g., to the stability analysis of asynchronous circuits, to intuitionistic (constructive) mathematical logic, and the existence of dynamic schedules (of the micro steps in a macro step).

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, 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 be constructively determined by the operational semantics of the programs. 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 still incomplete knowledge. This execution reveals the values of some further local/output variables of this macro step so that further micro steps can be executed. If all variables became finally known by this fixpoint iteration, the program is a constructive program with a unique behavior. For more details about causality analysis please refer to [29, 21, 11, 47, 8, 5, 39, 42].

3 Design Flow and Related Work

Before presenting the compilation procedure, we first introduce our overall design flow that determines the context of the compilation procedure presented in this report. As we target the design of embedded

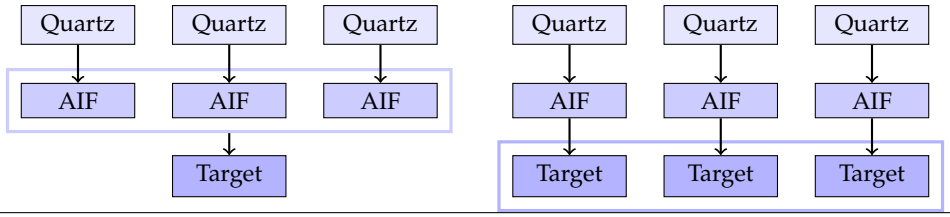

Figure 1: Modular Compilation and Modular Synthesis of Quartz Programs

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[1], which is a framework for the synthesis and verification of Quartz programs, we basically split up the design flow into two steps, which are bridged by the Averest Intermediate Format (AIF). This intermediate format models 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.

Fig. 1 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 runtime platform or the linker which have to organize the interaction of the target modules correctly.

Fig. 1 also makes clear that a *separate translation from source code modules to corresponding intermediate code modules* is required in both approaches. In this report, we will show that this step alone is not at all trivial since the separate translation has to consider potential preemption contexts of a module and potential surrounding loops that may lead to further schizophrenia problems that did not exist when compiling a module on its own. For example, assume that some (inner) module has been compiled to intermediate code, and it is later on used in another (outer) module. A first problem that might appear is that it can be called within a preemption statement so that its behavior must be adapted to the potential preemptions. Another problem might be that its outputs are bound to local variables in the outer module. This may lead to schizophrenia problems, which did not exist when the inner module was compiled. Similarly, additional problems arise when compiling the outer module without any information about termination or instantaneity of the inner one (which is desirable so that changes on inner modules do not imply recompilation of outer modules).

This report therefore focuses on the core ideas of our separate translation from Quartz modules to corresponding AIF modules [9, 38], which is the first one that supports the full modularity provided by the source language with unrestricted use of local variables, preemption statements, and delayed assignments. In our approach, additional information about the context of a module and a new handling of incarnations is provided to achieve a modular compilation of imperative synchronous programs. As Sections 7 and 8 show, this information is essential for binding. Otherwise it is impossible to determine when the guarded actions of a called module should be activated, aborted and suspended, or how many times a called module must be replicated due to reincarnations. This is also the reason why we do not use existing intermediate formats for synchronous languages as discussed in Sections 5.2, since none of them provides the required information for a separate translation.

Previous approaches like [27, 28, 33, 49, 50] have not considered these important problems with respect to modularity, since they either compiled synchronous *data-flow* languages that neither offer preemption statements nor dynamic incarnation of local variables, or they imposed the restriction that all called modules run in parallel [49] to each other to circumvent the mentioned problems concerning preemption and schizophrenia. Moreover, these publications mainly addressed the problem of modular causality analysis. To this end, one may impose certain restrictions like the use of inputs and outputs that are formulated with corresponding interfaces. Alternatively, one may perform a partial causality analysis on single modules. However, it still has to be reworked and completed at the end, since the

[1] `http://www.averest.org`

$a = 1;$	$b = a;$	$a = 1;$
$b = a;$	$a = 1;$	*if*$(b = 1)$ $b = a;$
pause;	*pause*;	*pause*;
$a = b;$	$a = b;$	*if*$(a \neq 2)$ $a = b;$
(a)	(b)	(c)

Figure 2: Three Quartz Programs Illustrating the Synchronous MoC

solution of causality problems requires information about all actions reading and writing to a variable that is only available at the end.

However, the mentioned publications on modular compilation do not propose solutions to the problems we mentioned. These problems have to be solved in the separate translation to the intermediate code as presented in this report. Causality problems, on the other hand, can also be solved later on: Since the source and intermediate code are both based on the synchronous MoC, they can model the same causality problems. For these reasons, we do not consider causality analysis in this report, although we also believe that a modular approach to causality analysis is desirable. However, we postpone it to a later compilation phase, and focus on the first one in this report

4 Quartz

In this report, we consider the synchronous language Quartz [38], which has been derived from Esterel [7, 6]. In the following, we give a brief overview of the language core, which is sufficient to define most other statements as simple syntactic sugar. For each statement, we will subsequently describe its behavior. For the sake of simplicity, we do not give a formal definition; the interested reader is referred to [38], which also provides a complete structural operational semantics. The Quartz core consists of the following statements, provided that S, S_1, and S_2 are also core statements, ℓ is a location variable, x and τ are a variable and an expression of the same type, σ is a Boolean expression, and α is a type:

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*)

In imperative synchronous languages, the synchronous MoC is represented as follows: All statements are assumed to be executed in zero-time. The only exception is a special statement *pause*, which implements the end of the current macro step. A simplified view on the programs is therefore as follows: In each macro step, a synchronous program resumes its execution at the *pause* statements where the control flow has been stopped at the end of the previous macro step, then it reads new inputs and executes the following statements until the next *pause* statements are reached. A *pause* statement also defines a control flow location implemented by a unique Boolean valued label ℓ, which is assumed to be true iff the control flow is currently at the statement ℓ : *pause*. Since all other statements are executed in zero time, the control flow can only rest at these positions in the program.

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, whereas delayed ones *next*$(x) = \tau$ transfer this value in the following macro step. If a variable is not set 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. Quartz

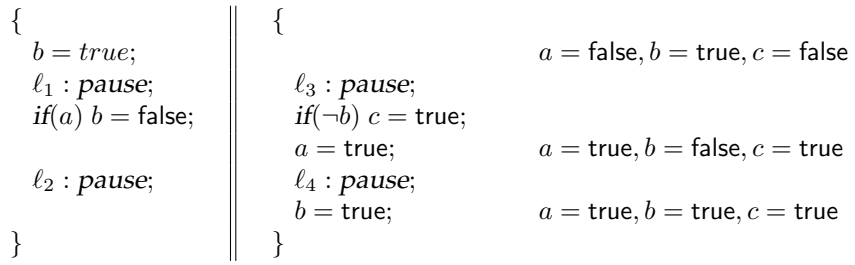

Figure 3: Synchronous Concurrency in Quartz

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 sets their values.

The assumption that all Quartz statements are executed in zero-time has some consequences which might be confusing at a first glance: if a statement does not take time for its execution, it is evaluated in the same variable environment as another statement following it in a sequence. In principle, both statements may therefore be interchanged without changing the behavior of the program. So, the program in Fig. 2 (b) has the same behavior as the program in Fig. 2 (a). Thus, each statement knows and depends on the results of all operations in the current macro step. Obviously, this generally leads to the causally cycles described in the previous section, which are not present in traditional sequential programming languages. Hence, a Quartz statement may influence its own activation condition (see the program in Fig. 2 (c)).

In addition to the usual control flow statements known from typical imperative languages (conditionals, sequences and iterations), Quartz offers synchronous concurrency. The *parallel statement* $S_1 \parallel S_2$ 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 one of the sub-statements is active.

Fig. 3 shows a simple example consisting of two parallel threads. In the first step, the program, and thus both threads, are started. The first thread executes the assignment to b and stops at location ℓ_1, while the second thread directly moves to location ℓ_3. In the second macro step, the program resumes are the labels ℓ_1 and ℓ_3. Since the second thread contains an immediate assignment to a, the action resetting b in the first threads is activated, which in turn activates the actions setting c in the second thread. The last step then resumes from ℓ_2 and ℓ_4, where the second thread performs the final assignment to variable b.

Preemption can be conveniently implemented by the *abort* and *suspend* statements. Their meaning 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 current data flow of this macro step. The immediate variants check for preemption already at starting time, while the default is to check preemption only after starting time.

Fig. 4 shows examples of all four abort variants. The execution of the first two statements requires two macro steps: in the first macro step, the abort block is entered without checking the condition, and a is set. In the second macro step, the condition is checked. While the strong variant immediately aborts and continues with the assignment to c, the weak one first completes the macro step by setting b and then

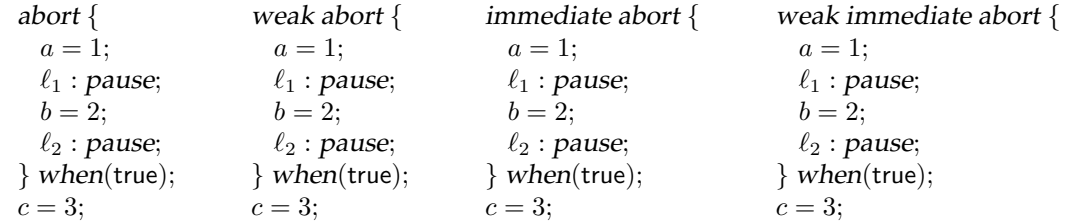

Figure 4: Abort Variants: Strong, Weak, Immediate Strong and Immediate Weak

moves to the assignment of c. The last two fragments feature immediate version of the abort statement. Since the abortion condition is checked when the control flow enters, both examples only need one step for their execution: in the strong immediate variant only c is set, while the weak immediate one additionally assigns a.

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.

5 Averest Intermediate Format (AIF)

5.1 Synchronous Guarded Actions

It is quite natural to divide the overall compilation and synthesis process into several steps. Especially in 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[2], which is a framework for the synthesis and verification of Quartz programs, we have chosen *synchronous guarded actions* to describe the system behavior in our intermediate format. They are in the spirit of traditional guarded commands [12, 15, 23, 24] but follow the synchronous MoC.

Hence, the behavior (control flow as well as data flow) is basically described by sets of guarded actions of the form $\langle \gamma \Rightarrow \mathcal{C} \rangle$. The Boolean condition γ is called the guard and $\mathcal{C}$ is called the action of the guarded action, which corresponds to an action of the source language. In this report, these are 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 a guarded action $\gamma \Rightarrow \mathcal{C}$ is that the action $\mathcal{C}$ is executed iff the condition γ is satisfied. Guarded actions may be viewed as a simple programming language like Unity [12] in that every guarded action runs as separate processes in parallel that observes the guard γ in each step, and executes $\mathcal{C}$ if the guard holds. The *semantics of synchronous guarded actions* is simply defined as follows: In each 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 in Quartz programs, there may be interdependencies between actions and trigger conditions so that any real implementation must execute the actions 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, if is defined according to the reaction to absence as described in Section 4.

AIF also makes it possible to overwrite the default *reaction to absence* for each variable, which is the key for the elimination of the schizophrenia problem in the intermediate code. Such an absence reaction for a variable x is stored in the triple (x, v_0, τ), where v_0 is the value in the initial macro step if no actions sets the value, and τ is an expression evaluated in the *preceding* step if no action sets the value in any later macro step. Hence, for a Boolean *event* variable x, we have the absence reaction $(x, \mathsf{false}, \mathsf{false})$, for a Boolean *memorized* variable x, we have the absence reaction (x, false, x). In principle, these customized absence reactions can also be expressed with the help of guarded actions and additional variables. However, as Section 8 will reveal, this destroys modularity, since additional guarded actions for a variable x change the guard of the reaction to absence of x. Therefore, AIF keeps the absence reactions separate from the guarded actions.

In addition to the description of the behavior by guarded actions and absence reactions, AIF contains more information to support modularity, which will be introduced in Section 8. Furthermore, other aspects in the design flow, such as specifications are covered in the intermediate format, which is however out of the scope of this report (and thereby not needed for its understanding).

[2] `http://www.averest.org`

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. On the one hand, problems like schizophrenia and the semantics of complex control flow statements like preemption statements can be completely solved during the translation to guarded actions, so that subsequent analysis, optimization and synthesis become much simpler. On the other hand, despite their very simple structure, efficient translation to both software and hardware is efficiently possible from guarded actions.

Guarded actions allow *many analyses and optimizations*. In particular, causality analysis can be effectively performed on guarded actions. If the causality analysis determined that a set of guarded actions does always have a dynamic schedule to compute the variables without first reading them in each macro step, then even an acyclic set of guarded actions can be determined. Other transformations on guarded actions are the grouping of guarded actions with regard to the variable they modify, which corresponds to the generation of static single-assignment form in the compilation of sequential languages. For synchronous languages this is often called an equational code generation, since for every variable, a single equation is generated.

5.2 Intermediate Code Representations of Synchronous Languages

There are several other intermediate formats which are used for the compilation of imperative synchronous programs [30, 20, 16, 32, 19, 43]. They are more or less related to our approach:

- *Imperative/Intermediate Code (IC):* The IC format is used by imperative languages like Esterel, Argos and Statecharts. It is the compilation result obtained after parsing, expanding macro statements, and type-checking. Hence, no essential code generation is performed, so that the format still offers concurrency, exceptions, local variables and calls to other modules. Compared to AIF, the format is close to the kernel Esterel language, and thus, it still contains much of the complexity of the source language.

- *Object Code (OC):* Object code, also called automaton code, is a low-level format that describes a finite state automaton by explicitly listing its states and transitions (and the code that is to be executed along the transitions). Hence, concurrency and complex interaction of threads has already been eliminated by the compiler. Object Code can be also created from guarded actions by reencoding the control states such that a single label (one-hot) represents each possible control flow position.

- *(Sorted) Sequential Circuit Code (SC and SSC):* These formats describe a hardware circuit that is obtained by compilation of Esterel programs. The unsorted version SC may contain combinatorial cycles, and the used gates are written in no particular order. In contrast, the SSC format makes use of a topological order and describes an acyclic circuit that may be obtained from a cyclic one by solving the corresponding causality problems. This intermediate format can be directly derived from guarded actions by grouping them according to their targets.

- *Declarative Code (DC):* The DC format is the privileged interchange format for hardware circuit synthesis, symbolic verification and optimization as well as distributed code generation. The format reflects declarative or data flow synchronous programs like Lustre, as well as the equational representations of imperative synchronous programs. The underlying idea is the definition of flows by equations governed by clock hierarchies. It is the successor of the previously used GC format [1] and is closely related to the sequential circuit code formats.

- *Event-Triggered Graphs:* Event-based simulation is a characteristic of hardware description languages like VHDL or Verilog. A compilation technique implemented in the Saxo-RT compiler of France Telecom [14, 13] also employed an event-triggered approach, and therefore relies on an internal data structure called an *event graph* that keeps track of dependencies of events. Hence, an event driven simulation scheme can be used to generate sequential code.

- *Concurrent Control Flow Graphs (CCFG):* The compiler of the University of Columbia translates Esterel programs to *concurrent control data flow graphs* [18, 19, 32, 16, 17]. At each instant, the control flow graph is traversed until nodes are reached that correspond with active control flow

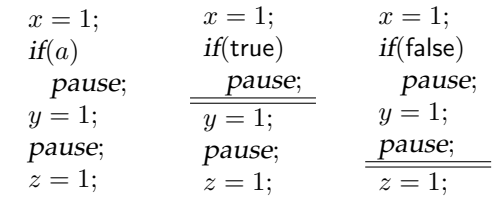

Figure 5: Surface and Depth

locations. Then, the corresponding subtrees are executed which changes the values of the control flow locations (that are maintained in Boolean variables). Hence, concurrency is still available, but sequential execution is also supported. Moreover, the format lends itself well for interpretation and software code generation for different architectures.

We decided against the usage of all these intermediate formats, since none of them contains the information needed for a separate compilation as described in the previous section.

6 Basic Compilation Scheme

In this section, we describe the basics of the translation from Quartz programs to guarded actions. For a better understanding, we first neglect local variables and module calls to obtain a simple, but incomplete translation procedure in this section. In Sections 7 and 8, we extend this procedure to local variables and module calls, respectively.

6.1 Surface and Depth

A key to the compilation of synchronous programs is their division into *surface and depth*. 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 *pause* statements. The depth contains the statements that are executed when the program resumes execution after a macro step, i.e., when the control is already inside the program and proceeds with its execution. It is important to see that surface and depth overlap, since *pause* statements may be conditionally executed. Consider the example in Fig. 5: 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 surface and depth of the sequence.

The example shown in Fig. 6 illustrates the necessity of distinguishing between surface and depth for the compilation. The compilation should compute for the data flow of a statement S guarded actions of the form $\langle \gamma \Rightarrow \mathcal{C} \rangle$, where $\mathcal{C}$ is a Quartz assignment, which is executed if and only if the condition γ holds. One might 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 correct, as the example on the left-hand side of Fig. 6 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 guards φ by $\varphi \wedge \neg i$ to implement the abortion. For the variable a, this incorrect approach would derive two guarded actions $\ell_0 \wedge \neg i \Rightarrow a = \mathsf{true}$

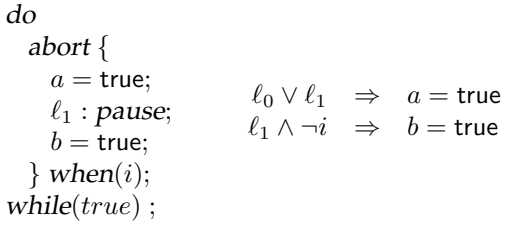

Figure 6: Using Surface and Depth for the Compilation

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.

So, what went wrong? The example shows that we have to distinguish between the guarded actions of the surface and the depth of a statement. If we could present these actions in two different sets, then we can 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.

In the following, we first present the compilation of the control flow in Section 6.2, before we focus on the data flow in Section 6.3. In both cases, we distinguish between the surface and the depth. As we will see at the end of that section, both parts can be integrated.

6.2 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 each label ℓ a set of conditions $\phi_1^\ell, \ldots, \phi_n^\ell$ under which this label will be activated 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, e. g. 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:

- strt_S is the current activation condition. It holds iff S is started in the current macro step.
- 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.
- susp_S similarly describes the suspension context: if the predicate holds, S will be suspended. Thereby, abrt_S has a higher priority, i.e., if both abrt_S and susp_S hold, then the abortion takes place.

The compilation of S returns the following control-flow predicates [36], which are used for compilation of surrounding compound statement.

- inst_S holds iff the execution of S is currently instantaneous. This condition depends on inputs so that we compute an expression inst_S depending on the current values of input, local and output variables. In general, 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.
- insd_S is the disjunction of the labels in statement S. Therefore, 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.
- 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 term_S therefore depends on input, local, output, and location variables. term_S is false whenever the control flow is currently not inside S. In particular, term_S is false for the instantaneous atomic statements.

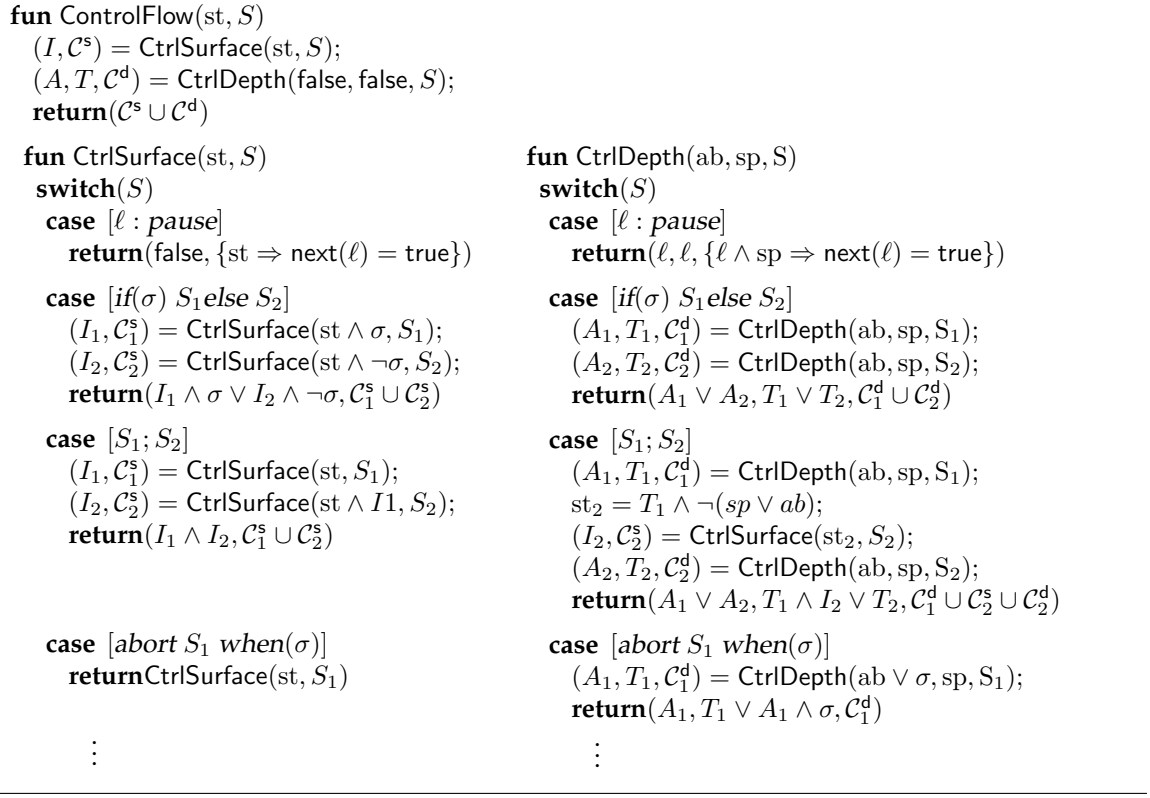

Figure 7: Compiling the Control Flow (Excerpt)

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 strt_S and inst_S, while the depth depends on abrt_S, susp_S, insd_S and 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. 7). 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 for the system.

It remains to show how the surface and the depth of each statement is 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 $\mathcal{C}$ is always instantaneous $\mathsf{inst}_\mathcal{C} = \mathsf{true}$, never active ($\mathsf{insd}_\mathcal{C} = \mathsf{false}$) and never terminates (since the control flow cannot rest inside $\mathcal{C}$, 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 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 in the very same situation ($\mathsf{term} = \ell$).

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 be also found in the functions for the compilation of the loops. Preemption is also

```
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) S_1 else S_2]
    D^s_1 = DataSurface(st ∧ σ, S_1)
    D^s_2 = DataSurface(st ∧ ¬σ, S_2)
    return(D^s_1 ∪ GC S_2)
  case [S_1; S_2]
    D^s_1 = DataSurface(st, S_1)
    D^s_2 = DataSurface(st ∧ inst_{S_1}, S_2)
    return(D^s_1 ∪ GC S_2)
  case [abort S_1 when(σ)]
    return DataSurface(st, S_1)
  case [weak abort S_1 when(σ)]
    return DataSurface(st, S_1)
    ⋮

fun DataDepth(S) =
 switch(S)
  case [x = τ]
    return{}
  case [next(x) = τ]
    return{}
  case [if(σ) S_1 else S_2]
    D^d_1 = DataDepth(S_1)
    D^d_2 = DataDepth(S_2)
    return(D^d_1 ∪ D^d_2)
  case [S_1; S_2]
    D^d_1 = DataDepth(S_1)
    D^s_2 = DataSurface(term_{S_1}, S_2)
    D^d_2 = DataDepth(S_2)
    return(D^d_1 ∪ D^s_2 ∪ D^d_2)
  case [abort S_1 when(σ)]
    return
      {γ ∧ ¬σ ⇒ C | (γ ⇒ C) ∈ DataDepth(S_1)}
  case [weak abort S_1 when(σ)]
    return DataDepth(S_1)
    ⋮
```

Figure 8: Compiling the Data Flow (Excerpt)

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. This concludes the compilation of the control flow.

6.3 Data Flow

Fig. 8 shows some of the functions which compute the data flow for a given Quartz statement. 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 since the precondition is the active current control flow location itself. The compilation of the surface of a basic statements 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.

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 simple the precondition 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

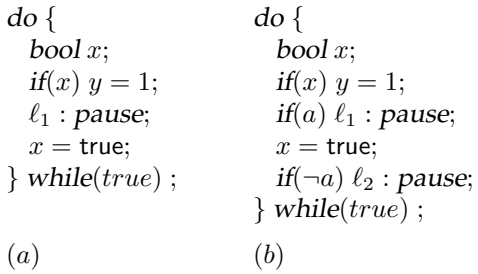

Figure 9: Schizophrenic Quartz Programs

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. 7) and the compilation of the data flow (see Fig. 8) can me 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 DataSurface(S,) runs in $O(|S|)$ and DataDepth(S) in $O(|S|^2)$. The reason for the quadratic blow up is that sequences and loops generate copies of surfaces of their substatements.

However, when combining control flow and data flow we must take care about the activation condition st. If we compare the variant used for the control flow in the previous section and the one used for the data flow in this section, we see that they do not match. The reason is that weak preemption statements distinguish between the *start of the depth* that allows the control flow to enter the statement and the *start of the surface* where the control flow will not enter the statement, but nevertheless some of its actions are enabled.

Hence, we have to distinguish them in the compilation by introducing an additional argument pr in the recursive call (see Fig. 11). Thereby, st enables the actions of the surface. If $\neg\mathrm{pr}$ holds, the control will additionally enter the considered statement, and if pr holds, a weak preemption takes place. For example, consider *weak abort* $S_1; S_2$ *when*(σ), and assume that σ holds and that the control is currently at a position in S_1 where S_1 terminates. As the abortion is weak, we have to execute the actions of the surface of S_2, but the control must not enter S_2. Hence, the start condition st of S_2 holds, and pr is true.

7 Local Variables

For a better understanding of the basic compilation principle, previous sections have neglected the declaration of local variables, which are known to be the source of subtle and challenging problems for compilers. In the following, we show how they are handled by our compiler.

7.1 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. This always occurs when local declarations are nested within loop statements. In such a problematic macro step, the micro steps must then refer to the right incarnation of the local variable, depending on whether they belong to the old or the new scope of the local declaration.

Fig. 9 (a) gives 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 software compilation of sequential programs can solve this problem simply by shadowing the incarnations of the old scope, this is not possible for the 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 (the old or the new one), the general case is more complicated as can be seen in Fig. 9 (b). 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 [5]. 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 Figure 10. The right-hand side of the same figure shows the corresponding graph. The circle nodes of this graph are control flow states that are labelled 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 towards 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 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 [31, 5, 36, 44, 48], which can handle more or less aspects of the problem (see [40] for more details). In the following, we present a general solution [9], which is tightly integrated in the compilation procedure presented in the previous section. For the sake of simplicity, we assume in the following that all variables have different names, hence, there is no shadowing.

7.2 Reincarnation Indices

Recall that the schizophrenia problem is due to an overlapped execution of possibly several surfaces and the depth of local declarations, since all these parts belong to different scopes of the local variable. Hence, the key to solve schizophrenia problems is again to compute control and data flow *separately for the surface and depth of a program*. Furthermore, we use an additional integer $\hbar$ for the compilation, which specifies the current incarnation level of the local variables. Using this integer $\hbar$, we can appropriately endow the variable occurrences with their incarnation indices, so that we can safely distinguish between the different scopes, when guarded actions of different surfaces are merged later on.

Furthermore, it is worth mentioning that it is sufficient to handle this issue during the compilation of loops (and not of local declarations!). Loops are responsible for the schizophrenia problem, since they are the only statements that allow to jump back in a program, which is required to reenter a scope. Hence, in the translation of loops (see Fig. 11), we must rename all local variables in the surface that is computed by Depth with the next incarnation level $\hbar + 1$. This level is then used when the actions are created to endow all variables with the appropriate index.

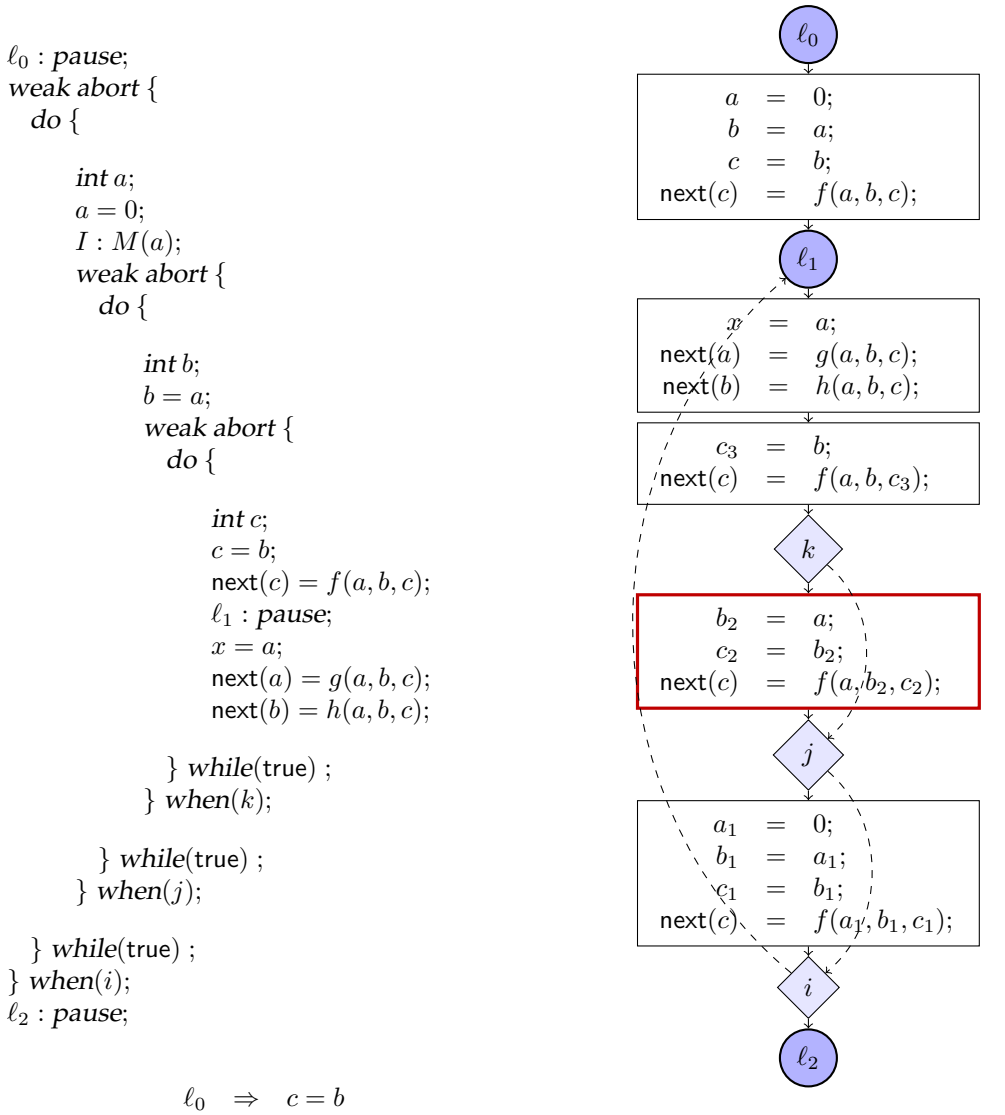

Figure 10: Challenging Local Declarations in a Quartz Program

```
fun Surface(ħ, st, pr, S)
 switch(S)
  case [{α x; S1}]
    (I, D^s, C^s, T) = Surface(ħ, st, pr, S1)
    if storage(α) = memorized then
      T' = T ∪ {(x, st ∧ ¬pr, x)}
    else T' = T
    return (D^s, C^s, T')
  case [do S1 while(σ)]
    return DataSurface(st, S1)
      ⋮
```

```
fun Depth(ħ, sp, ab, S)
 switch(S)
  case [{α x; S1}]
    (T, A, D^d, C^d, A) = Depth(ħ, sp, ab, S1)
    A' = A ∪ {x, {}, Default(α)}
    return(T, A, D^d, C^d, A')

  case [do S1 while(σ)]
    (T, A, D^d, C^d, A) = Depth(ħ + 1, sp, ab, S1)
    st = T ∧ σ
    pr = ab ∨ sp
    (I, D^s, C^s, T) = Surface(ħ, st, pr, S1)
    A' = A :: T
    T' = T ∧ ¬σ
    return (T', A, D^s ∪ D^d, C^s ∪ C^d, A')
      ⋮
```

Figure 11: Compiling Local Variables and Loops

7.3 Transfer Actions

Finally, the last thing that must be added to handle schizophrenia is to add actions that handle the transfer of the right incarnation to the following step. This transfer is necessary, since there may be several incarnations of a variable in the first step of a loop that all refer to the same local variable. In all other steps of the loop (in particular in the second one), these reincarnations do not exists anymore. Instead, there is only a single variable, which must be given the value of the *right* incarnation of the initial step. Thereby, the right incarnation is the one of the last (outermost) loop surface, which can only be determined dynamically, since it depends on the dynamic control-flow conditions.

Consider the example of Fig. 10 and assume that $\neg i \wedge \neg j \wedge k$ holds. Then, the outermost activated surface is the one of the second loop, which is marked with a red border in the flow diagram on the right-hand side. Thus, its reincarnations b_2 and c_2 must be transferred to the depth values of b and c. Without any look at the flow diagram, the outermost surface can be always found by evaluating their entering condition $\mathrm{st} \wedge \neg \mathrm{pr}$: only for the outermost one, it evaluates to true, since only its execution is followed in the next step.

To add the transfers, we determine the entering condition for the surfaces and add them as absence reactions to the intermediate code. Thus, if the variable of the depth is not explicitly set by an action, we transfer the old value of the right incarnation. In the compilation algorithm above, this information is computed by $\mathcal{T}$ and $\mathcal{A}$.

In $\mathcal{T}$, the enabling condition and the transfer for a particular surface is determined. Recall that reaction to absence is represented by triples (x, v_0, τ). To implement transfers, we use a *case* expression as default expression τ, which selects the right incarnation or another default value. Hence, all the parts are merged to a single *case* expression in $\mathcal{A}$, which adds each transfers as a separate case to the absence reaction (as indicated by ::).

In the lower left part of Fig. 10, the actions generated for the immediate assignment to c are listed (neglect the delayed ones until the next section). Since it can be reached in four surfaces (each one having a different incarnation level), four guarded actions with corresponding indices are created. On the right-hand side, one can notice the different incarnation indices of the variables. Finally, in the lower right part of the figure, the generated absence reaction of variable c is shown, which does a case distinction on the activation conditions of the contributing surfaces.

7.4 Late Actions

With the modifications presented so far, the compilation algorithm can handle schizophrenic variables with immediate actions. However, there is another problem related to local variables which is due to

delayed actions. Assume that a delayed action writing a local variable x is executed in the last step of the scope of x. Obviously, its effect, which becomes visible in the following step, should be ignored, since the scope has already been left. Hence, from the semantical point of view, these situations must be detected in the compilation algorithm, and the guards of the corresponding actions should be strengthened.

One might first think that is not necessary, since actions writing to variable x whose scope is left do not have any influence on the program behavior because there cannot be an expression reading x outside its scope. However, the problem is that the scope of x can be reentered in the following step, which then leads to the same confusion as schizophrenia in the previous section. Hence, the actions must be also deactivated to achieve correctness. The deactivation must be also done if the reason for the termination is a weak abortion, since the scope is left, and therefore, the next value of this incarnation is lost.

We must also disable delayed actions on local variables in those surfaces of local declarations that do not directly proceed to their depth. As already shown in the previous section, the surface of a local declaration can be executed more than once, but only one of these surface instances proceeds to the depth without leaving the scope. Only delayed actions of this instance of the surface are executed. For example, in Fig. 10, at most one of the actions $\mathsf{next}(c) = f(a, b, c)$, $\mathsf{next}(c) = f(a, b, c_3)$, $\mathsf{next}(c) = f(a, b_2, c_2)$, and $\mathsf{next}(c) = f(a_1, b_1, c_1)$ must be executed.

This disabling is done in the translation of local declarations, when the scope of a variable x is closed. We simply strengthen the guards of all delayed actions by the entering condition of the variable scope $\neg\mathrm{pr}$. This completes the modifications that must be done for the compilation of local variables. The guards of the generated guarded actions for the delayed assignment to c (shown in the lower left part of the Fig. 10) illustrate this deactivation: they are only fired if the corresponding surface is not weakly aborted, which guarantees that they are mutually exclusive.

8 Modules

Modules can be called at an arbitrary place in another module, since module calls are orthogonal to all other statements of the language. Thereby, the calling module defines the context of the called one. This view to module instantiation is different to the one data-flow languages take, where modules of systems all run in parallel and thus, in the same context.

Section 3 already outlined that each module is compiled to a corresponding module of the intermediate language. Hence, modular compilation means that we transform a set of source code files into a set of modules in the intermediate code. Thereby, two degrees of modularity must be distinguished. *Incremental compilation* requires that the compiled code for the inner module is available when the outer one is compiled. In this case, the compiler is able to simply include the already compiled code to the remaining part of the system. In contrast, *separate compilation* is more difficult and powerful: it allows one to compile a module without having any knowledge about the called modules except for their interfaces, since integration of called modules is deferred to a later linking step. Hence, modules can be compiled that call modules where only an interface exists. This allows one to change the implementation of a called module without the need to recompile the calling modules (only new linking is required). Clearly, this is a very important feature to efficiently create and maintain non-trivial systems consisting of many modules.

Unfortunately, none of the existing Esterel compilers supports a separate compilation. Separate compilation also has to handle preemption and schizophrenia outputs of the called module, which may be mapped to local variables in the calling module. The compilation procedure presented in the previous chapters must be adapted as described by the following subsections.

8.1 Module Context

In addition to the guarded actions of the module itself, a linked module also includes the guarded actions of the called modules. For separate compilation, the guarded actions of the called modules are not available so that the compiler can not yet simply include them. For this reason, our intermediate format provides sections where references to external module calls are collected (together with some context information).

As module calls may occur at arbitrary places in calling modules, the guards of their actions depend on the context of the calling modules: In particular, their guards have to respect activation or preemption

```
fun Surface(ħ, st, pr, S)
 switch(S)
  case [abort S1 when(σ)]
    return Surface(st, pr, S1,
)
  case [weak abort S1 when(σ)]
    return Surface(st, pr, S1,
)      ⋮
```

```
fun Depth(ħ, sp, ab, sg, S)
 switch(S)
  case [S1; S2]
    (T1, A1, D1d, C1d, A1) = Depth(ħ, sp, ab, sg, S1)
    (T2, A2, D2d, C2d, A2) = Depth(ħ, sp, ab, sg, S2)
    (I, Ds, Cs, T) = Surface(ħ, T1 ∧ ¬sg, sp ∨ ab, S2)
    return (T1 ∧ I, A1 ∨ A2, Ds ∪ D1d ∪ D2d, Cs ∪ C1d ∪ C2d, A1 ∪ A2)
  case [abort S1 when(σ)]
    (T, A, Dd, Cd, A) = Depth(ħ, sp, ab ∨ σ, sg, S1)
    return(T ∨ σ, A, Dd, Cd, A)
  case [weak abort S1 when(σ)]
    (T, A, Dd, Cd, A) = Depth(ħ, sp, ab ∨ σ, sg ∨ σ, S1)
    return (T ∨ σ, A, Dd, Cd, A)
      ⋮
```

Figure 12: Modular Compiling of Abort Statement

```
module M (int q){
  if(q < 10)
    next(q) = q + 1;
  else {
    ℓ1 : pause;
    q = 0;
  }
}
```

context: $\mathsf{inst}_M = (q < 10)$, $\mathsf{insd}_M = \ell_1$, $\mathsf{term}_M = \ell_1$

surface:
$$\left[\begin{array}{rcl} \neg\mathsf{prmt}_M \wedge \mathsf{strt}_M \wedge \neg(q<10) & \Rightarrow & \mathit{next}(\ell_1) = \mathsf{true} \\ \mathsf{strt}_M \wedge (q<10) & \Rightarrow & \mathit{next}(q) = q+1 \end{array}\right]$$

depth:
$$\left[\begin{array}{rcl} \ell_1 \wedge \mathsf{susp}_M & \Rightarrow & \mathit{next}(\ell_1) = \mathsf{true} \\ \ell_1 \wedge \neg\mathsf{strg}_M & \Rightarrow & q = 0 \end{array}\right]$$

Figure 13: Modular Compilation : Callee

conditions given by the context. This information must be passed to the module, and conversely, the module itself must provide status information (e. g. about its activation or termination) to the calling module.

Hence, in addition to the data variables exposed by the interface at the source code level, each AIF module M implicitly provides a *context interface*, which is designed in the style of the arguments of the recursive compilation procedures, which has been presented in the previous sections. Hence, modules can additionally use the following arguments: strt_M, prmt_M abrt_M, susp_M, inst_M, insd_M, term_M.

However, this extended set of arguments is not sufficient for separate compilation (but enough for incremental compilation) when compiling strong preemptions. While preemption of the surface part is directly handled by strengthening the condition strt_M, incremental compilation reprocesses the previously computed guarded actions each time a strong preemption context is compiled in the depth: the guard of each action is strengthened by the preemption guard, which requires that all guarded actions (including the ones from other modules) are known (see DataDepth(*abort* S *when*(σ)) in Fig. 8). Instead, the separate compilation introduces an additional context signal input $\mathrm{sg} = \mathrm{strg_M}$, which holds if the data flow of the depth must be preempted due to a surrounding strong abortion or suspension statement. This signal is then added to the guards of all actions of the module depth. Fig. 12 shows how this is implemented in the compilation procedure. The compilation of the strong abort adds the abortion condition σ to the context signal sg. The compilation of the sequences and loops later forwards this signals to its surfaces by using it as a part for the start signal st.

Recall that susp_M and abrt_M only affect the control flow and therefore, there is no distinction between strong or weak preemption. This missing information is provided by strg_M, which thereby disables the data flow in the depth. It does not need to distinguish between abortion and suspension so that we have the invariant $\mathsf{strg}_M \rightarrow (\mathsf{susp}_M \vee \mathsf{abrt}_M)$.

Fig. 13 shows an example. The module M shown on the left-hand side is compiled to the context information and actions shown on the right-hand side. One can notice how the context inputs are used

in the guards of all actions. In particular, note that the action $q = 0$ in the depth is strengthened by strg_M as described above.

8.2 Local Variables

Schizophrenia problems, which have been discussed in the previous section may lead to code duplication. This may even lead to duplication of module calls so that incarnations of module calls must also be distinguished in the intermediate format. Each copy of such a module call will thereby duplicate the actions of the called module. Fortunately, it is not necessary to duplicate the entire module. Instead, only a copy of its surface behavior (which is the only part of the module that can interfere with other incarnations of the same module) is sufficient. For this reason, we store the guarded actions of the *surface* and the *depth* part separately in our intermediate format.

In particular, it should allow us to handle the instantaneous reincarnation of local variable declarations in a modular way: This means that even though module calls inside loops may generate additional reincarnations of local variables, the linker does not have to take care about these reincarnations in an explicit way. Instead, the required reincarnations are implicitly made by generating copies of module calls in the different surfaces of the nested loops. Hence, a multi-dimensional numbering of the incarnations is needed: from the conceptual point, not single variables are duplicated, but instead, the surface of a module (including other surfaces) is duplicated with the potentially included copies of local declarations. To this end, the revised compiler makes use of qualified names, which are used to distinguish variable names that stem from the same name but are used in different module calls and incarnations.

For example, consider again the Quartz module shown in Fig. 13 and assume that it is called in the program shown in Fig. 10 at the indicated position. Since the call is schizophrenic, the outer module instantiated three references to M: one depth and two surfaces of M with context $\mathsf{strt}_I = w_0; \mathsf{prmt}_I = i$ and $\mathsf{strt}_{I_1} \ell_1 \wedge j, \mathsf{prmt}_{I_1} i$, which are bound to a and its reincarnation a_1, respectively. Furthermore, the labels ℓ_1 used in both modules must be distinguished by prefixing the one of the callee M by the instance name I (which would be also done for other local variables and module calls in M if they existed).

Finally, a very subtle problem related to late actions has to be solved, which is similar the one presented in the previous section. Output variables of a called module can be associated to local variables of the calling module, and therefore, a delayed assignment to an output variable of a module M may become effective after the scope of the bound local variable has been left. Hence, the delayed action must be deactivated in these cases. However, this deactivation cannot be made when compiling the called module, since it does not have any information about its callers. Similarly, this issue cannot be solved when compiling the calling module since it does not have access to the guarded actions of the called module.

The only possible solution is to defer the deactivation to the linker in these cases. The compiler creates the required information for this task by endowing all arguments of the module calls by appropriate (de)activation conditions (see previous section), which are finally added to the guards of the linked module.

Again, consider the two Quartz modules shown in Fig. 10 and Fig. 13. Since the inner variable q is bound to a local variable a of the caller, there is a two potential late action due to the delayed assignment to q in M (which is bound to a in the caller). Since the scope of a is left in the following step if the abortion condition j holds, the argument q is endowed with the deactivation condition j, which is finally added to the guards of all delayed assignments writing q.

The solution of this final issue concludes the separate compilation to the intermediate format.

9 Summary

In this report, we showed how imperative synchronous programs can be compiled separately to synchronous guarded actions. In particular, we have considered the compilation of locally declared variables which is difficult for synchronous languages due to the phenomenon of schizophrenia problems. Furthermore, we highlighted separate compilation. To this end, our compilation scheme is based on an intermediate code format that holds the information computed by the compiler that is later required by the linker. In principle, the reincarnation problem is handled by reincarnating the module calls with a sophisticated qualification of variable names so that the linker need not care about local variables.

References

[1] P. Aubry and T. Gautier. GC: the data-flow graph format of synchronous programming. *ACM SIGPLAN Notices*, 30(3):83–93, March 1995.

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

[3] G. Berry. A hardware implementation of pure Esterel. *Sadhana*, 17(1):95–130, March 1992.

[4] G. Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[5] G. Berry. The constructive semantics of pure Esterel. `http://www-sop.inria.fr/esterel.org/`, July 1999.

[6] G. Berry. The Esterel v5 language primer. `http://www-sop.inria.fr/esterel.org/`, July 2000.

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

[8] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, Institut National de Recherche en Informatique et en Automatique (INRIA), Sophia Antipolis, France, September 1998.

[9] 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.

[10] J. Brandt, K. Schneider, and A. Willenbücher. Using IP cores in synchronous languages. In C. Gremzow and N. Moser, editors, *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV)*, pages 97–106, Berlin, Germany, 2009. Universitätsbibliothek Berlin, Germany. ISBN 9783798321182.

[11] J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, 1995.

[12] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, USA, May 1989.

[13] E. Closse, M. Poize, J. Pulou, J. Sifakis, P. Venter, D. Weil, and S. Yovine. TAXYS: A tool for the development and verification of real-time embedded systems. In G. Berry, H. Comon, and A. Finkel, editors, *Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 391–395, Paris, France, 2001. Springer.

[14] E. Closse, M. Poize, J. Pulou, P. Venier, and D. Weil. SAXO-RT: Interpreting Esterel semantics on a sequential execution structure. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5):80–94, 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[15] D.L. Dill. The Murphi verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, New Jersey, USA, 1996. Springer.

[16] S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (T-CAD)*, 21(2):169–183, February 2002.

[17] S. Edwards. ESUIF: An open Esterel compiler. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5):79, 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).

[18] S.A. Edwards. Compiling Esterel into sequential code. In *Hardware-Software Codesign (CODES)*, pages 147–151, Rome, Italy, 1999. ACM.

[19] S.A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):117–131, 2006.

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

[21] N. Halbwachs and F. Maraninchi. On the symbolic analysis of combinational loops in circuits and synchronous programs. In *Euromicro Conference*, Como, Italy, 1995. IEEE Computer Society.

[22] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(4):293–333, 1996.

[23] H. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.

[24] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Cooperation, 1991.

[25] Y.-T.S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer, 1999.

[26] G. Logothetis and K. Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Design, Automation and Test in Europe (DATE)*, pages 10196–10203, Munich, Germany, 2003. IEEE Computer Society.

[27] R. Lublinerman, C. Szegedy, and S. Tripakis. Modular code generation from synchronous block diagrams: modularity vs. code size. In Z. Shao and B.C. Pierce, editors, *Principles of Programming Languages (POPL)*, pages 78–89, Savannah, Georgia, USA, 2009. ACM.

[28] R. Lublinerman and S. Tripakis. Modularity vs. reusability: Code generation from synchronous block diagrams. In *Design, Automation and Test in Europe (DATE)*, pages 1504–1509, Munich, Germany, 2008. IEEE Computer Society.

[29] 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.

[30] J.-P. Paris, G. Berry, F. Mignard, P. Couronne, P. Caspi, N. Halbwachs, Y. Sorel, A. Benveniste, T. Gautier, P. Le Guernic, F. Dupont, and C. Le Maire. The common format of synchronous languages: The declarative code DC, 1998.

[31] A. Poigné and L. Holenderski. Boolean automata for implementing pure Esterel. Arbeitspapiere 964, GMD, Sankt Augustin, Germany, 1995.

[32] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 227–236, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[33] M. Pouzet and P. Raymond. Modular static scheduling of synchronous data-flow networks: an efficient symbolic representation. In S. Chakraborty and N. Halbwachs, editors, *Embedded Software (EMSOFT)*, pages 215–224, Grenoble, France, 2009. ACM.

[34] F. Rocheteau and N. Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Algorithms and Parallel VLSI Architectures II*, Bonas, France, 1991.

[35] 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.

[36] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Application of Concurrency to System Design (ACSD)*, pages 143–154, Newcastle upon Tyne, UK, 2001. IEEE Computer Society.

[37] K. Schneider. Proving the equivalence of microstep and macrostep semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOL)*, volume 2410 of *LNCS*, pages 314–331, Hampton, Virginia, USA, 2002. Springer.

[38] K. Schneider. The synchronous programming language Quartz. Internal Report 375, Department of Computer Science, University of Kaiserslautern, Kaiserslautern, Germany, December 2009.

[39] 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, DC, USA, 2004. ACM.

[40] 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.

[41] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, pages 1–19, Edinburgh, UK, 2005.

[42] 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, St. Malo, France, 2005. IEEE Computer Society.

[43] K. Schneider, J. Brandt, and E. Vecchié. Efficient code generation from synchronous programs. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 165–174, Napa, California, USA, 2006. IEEE Computer Society.

[44] 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.

[45] T. Schuele and K. Schneider. Exact runtime analysis using automata-based symbolic simulation. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 153–162, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[46] T. Schuele and K. Schneider. Abstraction of assembler programs for symbolic worst case execution time analysis. In S. Malik, L. Fix, and A.B. Kahng, editors, *Design Automation Conference (DAC)*, pages 107–112, San Diego, California, USA, 2004. ACM.

[47] 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.

[48] 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.

[49] J. Zeng and S.A. Edwards. Separate compilation for synchronous modules. In L.T.Yang, X. Zhou, W. Zhao, Z. Wu, Y. Zhu, and M. Lin, editors, *International Conference on Embedded Software and Systems (ICESS)*, volume 3820 of *LNCS*, pages 129–140, Xi'an, China, 2005. Springer.

[50] Y. Zhou and E.A. Lee. A causality interface for deadlock analysis in dataflow. In S.L. Min and W. Yi, editors, *Embedded Software (EMSOFT)*, pages 44–52, Seoul, South Korea, 2006. ACM.