

# Separate Compilation for Synchronous Programs

Jens Brandt and Klaus Schneider

Embedded Systems Group  
Department of Computer Science  
University of Kaiserslautern  
<http://es.cs.uni-kl.de/>

**Abstract**—Esterel and other imperative synchronous languages offer a rich set of statements, which can be used to conveniently describe complex control behaviors in a concise, but yet precise way. In particular, the ability to arbitrarily nest all kinds of statements including loops, local declarations, sequential and parallel control, as well as several kinds of preemption statements leads to a powerful programming language. However, this orthogonal design imposes difficult problems for the modular or separate compilation, which has to deal with special problems like the instantaneous reincarnation of locally declared variables.

This paper presents a compilation procedure allowing to separately compile modules of a synchronous language. Our approach is based on two new achievements: First, we derive the information that is required for a linker to combine already compiled modules. This information is stored in a file written in an intermediate format, which is the target of our compilation procedure and the source of the linker. Second, we describe a compilation procedure for a typical imperative synchronous language to generate this intermediate format. We have implemented the approach in the upcoming version 2.0 of our Averest system<sup>1,2</sup>.

## I. INTRODUCTION

Synchronous languages [2] like Esterel [5] and its variants [20], [27] offer many important features to fulfill the requirements imposed by embedded systems: First, it is possible to generate efficient software *and* hardware from the same synchronous program [4], which allows a fast simulation of the application-specific hardware as well as late changes of the hardware-software partitioning of a system. Second, it is possible to determine tight bounds on the reaction time by a simplified worst-case execution time analysis (since loops do not appear in reaction steps). Third, the formal semantics of these languages allows the application of formal methods to prove (1) the correctness of the compilation and (2) the correctness of a particular program with respect to given formal specifications [24], [25], [26], [30].

However, the compilation of synchronous languages is much 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 without the help of an operating system for managing different processes or threads. To this end, special problems like causality and schizophrenia problems have to be solved by the compilers [6], [29], [30].

It is therefore reasonable and natural to split the overall compilation process into several phases, such that the first

steps are independent of the final target code. For obvious reasons, it is desirable to store the intermediate results so that they can be reused for further compilation: First, expensive optimizations may have already been performed on the intermediate results. In particular, complex statements like the interaction of concurrent threads with abortion and suspension statements as well as instantaneous broadcast communication can already be reduced to simpler statements in a first compilation phase. Second, the intermediate code can be distributed in libraries without revealing its proprietary source (which leads to the ideas of IP-blocks used in hardware design).

Before we can present our contribution to modular compilation of synchronous programs, we need to make our terminology more precise: We refer to *compilation* as the translation of source code into *target-independent intermediate code*, while *synthesis* means the translation from the intermediate format to the final target code. Hence, modular compilation means that we are able to transform a set of source code files into a set of modules in the intermediate code, which can be subsequently *linked* before the final synthesis takes place. It may also be possible that the intermediate code modules are individually synthesized and then linked, which is *modular synthesis*. Thus, modular compilation is a necessary precondition for modular synthesis — but not sufficient: further preconditions and restrictions are given by the specific target code. In this paper, we only consider modular compilation.

The term *module* is also a very fuzzy term which we have to make more precise. In our case, modules are provided by module definitions of the source language, which are compiled to corresponding modules of the intermediate language. Modules can be used by ‘calling’ (or better instantiating) them at an arbitrary place in the program. Restrictions of the use of already defined modules lead to different forms of modularity:

- The first view, which is used by most hardware description languages, considers modules as independent components that communicate over their interfaces. Most of these languages explicitly distinguish between behavior and structure. Modules are defined by the behavioral part of the language, and they are subsequently assembled by the structural part. Although there might be a structural hierarchy (which does not change over runtime), modules of systems created by this approach all run in parallel and thus, in the same context. As there are no context statements that could preempt or restart the behavior, this kind of symmetric linking is very simple. VHDL, SDL

<sup>1</sup>© 2009 by EDAA.

<sup>2</sup>See <http://www.averest.org>.

or even threads in all classical imperative programming languages follow this simple, but limited, approach.

- The second approach, which is followed by imperative synchronous languages, assumes that module instantiations are orthogonal to all other statements of the language. Thus, module instantiations can be used everywhere in the synchronous program. This gives the developer a higher level of abstraction, since sophisticated control flow as sequential/parallel execution or preemption is directly modeled in the program. A basic precondition for this approach is that linking is asymmetric since if one module instantiates another module, then the calling module defines the context of the called one. This is the view classic software languages usually have: A module is ‘called’ from another one.<sup>3</sup>

For the latter case, two degrees of modularity can be distinguished, which we call *incremental* and *separate* compilation:

- *Incremental compilation* requires that the compiled code for the inner module is available when the outer one is compiled. Then, the compiler is able to *simply include the already compiled code* to the remaining part of the system. All compilation procedures like [6], [30], [33] that are defined by a single recursive traversal over the syntax tree of the program can be easily generalized to implement incremental compilation by (1) simply storing all the compilation results of a compiled module in a file and (2) importing these results whenever a module instantiation of this module is compiled.
- In contrast, *separate compilation* is much more complex: It allows one to compile a module without having any knowledge about the called modules except for their interfaces, since *inclusion of called modules is deferred to a later linking step*. Hence, even modules can be compiled that call modules that are not yet implemented. This allows developers to compile all modules of the system in an arbitrary order and to link them at the end of the development process. Furthermore, separate compilation allows one to change a called module without the need to recompile the modules calling it (only new linking is required). Clearly, this is a very important feature to efficiently create and maintain systems consisting of many modules.

This paper presents the first algorithm that supports fully separate compilation for a typical imperative synchronous language like Esterel: Modules of the source language are translated to individual modules in an intermediate format, which does not require any information about the called modules except for their names and the types of their inputs and outputs. Our contribution is based on two new developments: First, we define a new intermediate code format called AIF (Averest Intermediate Format), which is a target-independent representation of the system in terms of simple guarded actions. Second, we develop a compilation procedure

<sup>3</sup>Obviously, the second approach comprises the first one: If two modules are run in parallel, the description of their connection can be seen as a separate module consisting of parallel module calls.

to generate this AIF code. Both developments are available in the upcoming version 2.0 of our Averest system.

A particular advantage of our separate compilation is that it also handles 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 compiler 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. This solution requires a sophisticated generation of names and related data structures in the intermediate code.

Our contribution is therefore quite different to previous work in this area, especially different to [37] which has a similar title. [37] ignores schizophrenia problems and the asymmetric relationship between called modules and focuses on causality problems instead<sup>4</sup>. If causality analysis is integrated to the modular compilation, an explicit causality interface like the one presented in [38] could be provided. We do not make use of such interfaces, since this would restrict our programs to acyclic dependencies.

This paper is structured as follows: In Section II, we give a brief overview over the synchronous language Quartz considered in this paper and the problems that have to be solved by compiling these programs. Section III presents our intermediate format, which must be provided by all components to be integrated by the linker (that is used for both hardware and software synthesis). Subsequently, Section IV explains how modules are compiled to intermediate code. Thereby, we focus on the problems that must be solved with regard to modular compilation. In Section V, we illustrate our approach by a small example before we draw some conclusions in Section VI.

## II. SYNCHRONOUS LANGUAGES

### A. Syntax and Semantics

Quartz [27] is an imperative synchronous language derived from the Esterel language [8], [7]. The common paradigm of all synchronous languages is perfect synchrony [17], [2] which means that the execution of a program is divided into micro and macro steps. The execution of micro steps is done in zero time, while the execution of a macro step requires one logical unit of time. Variables are constant during the execution of micro steps and synchronously change at macro steps. As from the programmer’s point of view, all macro steps take the same amount of logical time, concurrent threads run in lockstep and automatically synchronize at macro 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], [23], [27]. Furthermore, the concise formal

<sup>4</sup>From our point of view, causality analysis is not part of the compilation, but part of the linking, since causality checking can only be done in the complete system and not yet in its individual components.

semantics of synchronous languages makes them particularly attractive for reasoning about program properties and equivalences. Therefore, synchronous languages are well-suited for the design of safety-critical embedded systems that consist of application-specific hardware and software.

In the following, we give a brief overview of the Quartz language core, which is powerful enough to define most other statements as simple syntactic sugar. Due to lack of space, we do not describe the semantics of Quartz in detail, and refer instead to [27] and to the Esterel primer [7], which is an excellent introduction to synchronous programming. The Quartz core consists of the following statements, provided that  $S$ ,  $S_1$ , and  $S_2$  are also core statements,  $\ell$  is a location variable,  $x$  is a variable,  $\sigma$  is a Boolean expression, and  $\alpha$  is a type:

- `nothing` (empty statement)
- $x = \tau$  and `next(x) =  $\tau$`  (assignments)
- $\ell : \text{pause}$  (start/end of macro step)
- `if( $\sigma$ )  $S_1$  else  $S_2$`  (conditional)
- $S_1; S_2$  (sequential composition)
- `do  $S$  while( $\sigma$ )` (iteration)
- $S_1 \parallel S_2$  (synchronous concurrency)
- `[weak] abort  $S$  when [immediate]( $\sigma$ )`
- `[weak] suspend  $S$  when [immediate]( $\sigma$ )`
- $\{\alpha x; S\}$  (local variable  $x$  with type  $\alpha$  and scope  $S$ )
- `inst : name( $\tau_1, \dots, \tau_n$ )` (call of module *name*)

There are two kinds of assignments: Both immediately evaluate the right-hand side expression  $\tau$  in the current environment/macro step. Immediate assignments  $x = \tau$  instantaneously transfer the obtained value of  $\tau$  to the left-hand side  $x$ , whereas delayed ones `next(x) =  $\tau$`  transfer this value in the following macro step. Hence, the variable environment of a macro step that executes  $x = \tau$  must already satisfy the equation  $x = \tau$  (which may lead to cyclic dependencies considered by causality analysis).

There is essentially only one basic statement that defines a control flow location, namely the `pause` statement. These statements are endowed with unique Boolean valued *control flow location variables*  $\ell$ , which are true iff the control flow is currently at the statement  $\ell : \text{pause}$ . Since all other statements are executed in zero time, the control flow can only rest at these positions in the program.

In addition to the usual control flow constructs known from all kinds of imperative languages (conditionals, sequences and iterations), Quartz offers synchronous concurrency and preemption statements: 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 the following `pause` statement. The parallel statement runs as long as one of the sub-statements is active. The meaning of *preemption statements* is as follows: A statement  $S$  which is enclosed by an `abort` block is immediately terminated when the given condition  $\sigma$  holds. Similarly, the `suspend` statement freezes the control flow in a statement  $S$  when  $\sigma$  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. 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 first by the declaration of modules in the source code and second by calling 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. Thus, in addition to parallel execution (which is only allowed in HDLs), a module instantiation can also be part of sequences or conditionals. Furthermore, it can be located in any abortion or suspension context, which possibly preempts its execution. Hence, a compiler that generates code for these modules is obviously challenged by this flexibility.

### B. Compile Problems of Synchronous Languages

On the one hand, perfect synchrony simplifies concurrent programming, since developers do not have to bother about many low-level details like timing, synchronization and scheduling. On the other hand, this paradigm has some consequences that make the compilation of synchronous programs not at all straightforward. In particular, causality and schizophrenia problems challenge compilers.

*Causality problems* [29], [31], [28], [35] arise if statements have cyclic dependencies, i.e. when an action affects its own trigger condition like in the simple program `if(!x) x = true`. Since the condition and the assignment are executed in the same macro step (i.e. in the same variable environment), this statement has no consistent behavior: Assuming  $x$  is true, i.e., `!x` is false, then `x = true` is not executed, which gives us no reason<sup>5</sup> to assume that  $x$  should be true. Alternatively, assuming  $x$  is false makes `!x` true, and hence, `x = true` is executed, which which contradicts the assumption. Hence, this program has no consistent behavior. For this reason, compilers have to perform a causality analysis, which efficiently determines whether a program has a unique behavior that can be constructively determined. In general, even for causally correct programs dynamic schedules are required for their execution.

*Schizophrenia problems* [30], [36] arise from the fact that each variable has a unique value in each macro step. This is a problem for local variables whose scope is left and reentered within a single macro step. Clearly, this can only happen if the local declaration is nested in a loop. As the values of the old and the new scope can be different, local variables must be consequently duplicated in these situations. Hence, when entering a loop body, compilers must create new so-called reincarnations of the variables that are locally declared in a loop body. Since loops can be nested, a locally declared variable may have several reincarnations and the compiler has to keep track of which reincarnation finally propagates its value to the actual local variable.

## III. THE INTERMEDIATE FORMAT

As already outlined in the introduction, it is quite natural to divide the overall compilation and synthesis process into several

<sup>5</sup>We assume here that there is no context statement that could make  $x$  true.

steps. Due to the use 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. The synchronous languages community has already defined various intermediate formats [21], [17], [13], [22], [16], [32] including the following ones:

- *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, and even calls to other modules. The format is therefore close to the kernel Esterel language, and thus, it is still a high-level format.
- *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.
- *(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.
- *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 [11], [10] 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* [15], [16], [22], [13], [14]. At each instant, the control flow graph is traversed until nodes are reached that correspond with active control flow 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 se-

quential execution is also supported. Moreover, the format lends itself well for interpretation and software code generation for different architectures.

#### A. Guarded Actions

Instead of using one of the existing code formats mentioned above, we chose *guarded actions* [9], [12], [18], [19], a well-established concept for the description of concurrent systems, as the basis for our intermediate format. This choice is motivated by the following arguments:

- We are convinced that guarded actions are exactly at *the right level of abstraction* for an intermediate code format, since guarded actions provide a good balance between (1) removal of complex statements available at the source code level and (2) the independence of a specific synthesis target: On the one hand, complex control-flow statements like preemption statements have been eliminated during the translation to guarded actions. Hence, synthesis tools do not have to deal with the complex semantics of difficult (preemption) statements, and instead only have to cope with simple, flat guarded actions. Despite this very simple structure, efficient translation to both software and hardware is possible: OC and SC (which can be both directly derived from guarded actions) bear witness to this. Many intermediate formats do not have this property: whereas IC and CCFG are quite high-level and complex formats for simple synthesis back-ends, OC and SC are target-specific.
- Guarded actions allow *many analyses and optimizations*. In particular, causality analysis can be performed on guarded actions. If the causality analysis determined that a set of guarded actions does always have a dynamic schedule to compute the output 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. Note that this transformation does also yield equational code.
- Guarded actions have already been used in the compilation workflow of synchronous programs. For example, they are an explicit intermediate step in the translation to circuit code [30]. Hence, our new compilation scheme can benefit from previous results in this area.

Hence, the modules we consider for integration are basically defined by sets of guarded actions of the form  $\gamma \Rightarrow \mathcal{A}$ . The Boolean condition  $\gamma$  is called the guard and  $\mathcal{A}$  is called the action of the guarded action, which corresponds to an action of the source language. In this paper, these are the assignments of Quartz, i.e. the guarded actions have either the form  $\gamma \Rightarrow x = \tau$  (for an immediate assignment) or  $\gamma \Rightarrow \text{next}(x) = \tau$  (for a delayed assignment). The *semantics of these 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.

*Guarded actions are generated for the control and data flow of the program.* The data flow consists of all assignments of the program and determines the values of all declared variables. The control flow consists of guards  $\gamma \Rightarrow \text{next}(\ell) = \text{true}$  where  $\gamma$  is a condition that is responsible for moving the control flow at the next point of time to location  $\ell$ .

This behavior of a system additionally includes implicit guarded actions due to the so-called *reaction to absence*. It defines the value of a variable if no action has determined its value 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). Hence, our intermediate format additionally defines a default reaction for each variable depending on its declared storage typed: *Event variables* are reset to their default values, while *memorized variables* maintain their previous values.

In principle, the absence reactions can also be expressed as guarded actions. However, this can only be done at the final linking step, since adding additional guarded actions that modify a variable changes the guard of the reaction to absence of this variable. Hence, the reaction to absence can only be determined during the final linking phase. Nevertheless, we can compute its guard in a modular manner, so that the final linker just has to compute the conjunctions of the modularly computed guards. As the guards of these absence reactions have to be modified by the linker, the absence reactions must be kept separately from other actions.

### B. Module Instantiations

In addition to the guarded actions of the module itself, the linked module must also include the guarded actions of the called module instances. For a separate compilation, the called modules might not yet be available, so that the compiler can not yet simply include the guarded actions of called modules. For this reason, our intermediate format must provide sections where references to external module calls are collected together with some context information. Due to schizophrenia problems, which have been sketched above, it is possible that more than one reference is created in the intermediate format for a single module call in the source code. 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 initial 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 split the guarded actions of a module into a *surface* and a *depth* part [6], [34]: the surface of a module consist of those micro steps that are executed at initial time before the first control flow location is reached, while the depth contains the remaining actions of the module<sup>6</sup>.

Due to this distinction, the surface of a module may contain many references to surfaces of called modules, but at most

<sup>6</sup>It should be noted that some statements of the source program can belong to the surface and the depth.

one reference for each module instantiation. In contrast, the depth of a module may contain many surfaces that stem from one module instantiation in addition to its depth. Hence, the surface has a linear growth, while the depth has a quadratic one in terms of the size of the input program.

### C. The Context of Modules

Roughly speaking, linking several modules only consists of collecting their guarded actions. However, as module instantiations 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 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 the following *context interface*, which is again divided into two parts which affect the surface and the depth of the module. The surface has the following context:

- The  $\text{go}^s(M)$  input triggers the execution of the data flow actions in the surface of the module; no program location is affected by this input.
- The  $\text{go}^d(M)$  input is responsible for initially activating program locations inside a module. As we maintain the invariant  $\text{go}^d(M) \rightarrow \text{go}^s(M)$ , the entering of a statement by the control flow implies the execution of its surface. Once the control flow entered program locations inside the statement, these will enable the further execution in the following macro steps.
- The output  $\text{inst}(M)$  is true iff the module is currently instantaneous, i. e. if started now, its execution would only consist of micro steps (this depends on the current variables' values).

$\text{go}^s(M)$  always implies  $\text{go}^d(M)$ , and  $\text{go}^s(M) \wedge \neg\text{go}^d(M)$  holds iff the statement is weakly pre-empted. The context of the depth has the following interface:

- The input  $\text{abrt}(M)$  holds if the context contains an abortion statement whose condition holds. This input does not distinguish between weak and strong abortion (which is relevant only for the data flow), since it is used only for the definition of the control flow of the module.
- The input  $\text{susp}(M)$  holds if the context contains a suspension statement whose condition holds. Analogous to  $\text{abrt}(M)$ , this input does not distinguish between weak and strong suspension (which is also relevant only for the data flow), since it is used only for the definition of the control flow of the module.
- The input  $\text{prmt}(M)$  holds if the data flow of the depth of a statement must be preempted due to a surrounding strong abortion or suspension statement. Note that both  $\text{abrt}(M)$  and  $\text{susp}(M)$  only affect the control flow of a module, and preemption of the surface part is directly handled by the difference of  $\text{go}^s(M)$  and  $\text{go}^d(M)$ . However, a strong preemption of actions in the depth is recognized by the  $\text{prmt}(M)$  input only.

- The output  $\text{insd}(M)$  indicates whether a control flow in the module is still active, i.e. whether the module has been started in a previous cycle ( $\text{go}^d(M) = \text{true}$ ) and its computations have not yet terminated.
- The output term  $(M)$  is true iff  $\text{insd}(M)$  holds and the module currently terminates its computation on its own (i.e. without being aborted).

Recall that  $\text{susp}(M)$  and  $\text{abrt}(M)$  only affect the control flow and therefore, there is no distinction between strong or weak preemption, while  $\text{prmt}(M)$  is used to disable the data flow in the depth and therefore only considers strong preemption without further distinguishing between abortion and suspension. Thus, we have the invariant  $\text{prmt}(M) \rightarrow (\text{susp}(M) \vee \text{abrt}(M))$ .

#### D. AIF at a Glance

To summarize, the intermediate format has the following structure<sup>7</sup>:

- *name* (module name)
- *interface* (list of variables exposed at the data interface)
- *locals* (list of locally declared variables)
- *surface*
  - *context* (surface context)
  - *surfaceCalls*
  - *ctrlFlow* (guarded actions of the control flow)
  - *dataFlow* (guarded actions of the data flow)
- *depth*
  - *context* (depth context)
  - *surfaceCalls*
  - *depthCalls*
  - *ctrlFlow* (guarded actions of the control flow)
  - *dataFlow* (guarded actions of the data flow)
- *absReacts* (absence reactions)

The Averest system defines a scheme for a Extensible Markup Language (XML) file, which is used as the concrete syntax.

### IV. SEPARATE COMPILATION

The translation of synchronous programs to guarded commands is already a well-established procedure. The previous compiler of the Averest system used it as an intermediate step for the generation of equations or transition systems. We refer to [30] for a detailed presentation of the previous translation.

#### A. Adding Modularity

However, [30] (as all other documented compilers) does not support modular compilation: Some procedures of previous compilers made use of a global view of the system and considered information about the entire system. For example, for each local variable declaration, it is counted how many loops are nested around it, so that the number of required reincarnations could be easily determined. In a modular compilation, this information is no longer available, and therefore the compilation procedure must be fundamentally changed:

<sup>7</sup>The AIF format additionally contains specifications, which is neglected in this paper for the sake of simplicity.

- The *deactivation of the data flow*, which is necessary when a strong preemption takes place, cannot be implemented by reprocessing the previously computed guarded actions. Previously, this was done each time a strong preemption context was compiled: the guard of each action was strengthened by the preemption guard (which requires that all guarded actions including the ones from other modules, are known). Instead, the new compiler uses the context signal input  $\text{prmt}(M)$ , which is implicitly added to all guards of the module. As already outlined above, this signal only affects the depth of the module. Strong preemption in the surface is handled by the input  $\text{go}^s(M)$ .
- Although the guarded actions were computed in a recursive traversal over the syntax tree of the source program, the *resolution of schizophrenia problems* was deferred to the very end. All incarnations of a module were counted and afterwards, new incarnations are created depending on this index. However, in the modular case, the number of incarnations for any other module is not accessible. 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, our new compiler makes use of qualified names (QNames), which are used to distinguish variable names that stem from the same name but are used in different module calls and incarnations.
- A very tricky problem related to schizophrenia is posed by *delayed assignments which are executed in the last step in the scope of the local variable*. These updates must be always deactivated, since the scope of the variable is left. If the scope would be instantaneously reentered, the delayed assignment of the previous scope would interfere with the current scope, which would be wrong. This problem has already been solved for the non-modular case, but the modular case introduces a new technical challenge, since output variables of a module can be associated to local variables of the calling module: *A delayed assignment to an output variable of a module  $M$  must be deactivated if this variable is associated with a local variable by a module call to  $M$  if at this point of time the scope of the local variable is left*. Hence, this deactivation cannot be made when compiling the called module. However, also the calling module does not yet know the guarded actions of the inner module. Hence, the deactivation must be deferred to the linker, but the compiler has to endow all argument expressions of the module calls by appropriate (de)activation conditions (see previous section), which are finally added to the guards of the linked module.

#### B. Compiler Structure

The main idea is to compute in one pass over a given statement  $S$  the required information like the context information and the guarded actions of the surface and the depth. To this end, the translation has to forward start conditions  $\text{go}^s$  and  $\text{go}^d$  during

the recursive descent. Additionally, the translation maintains the conditions *ab*, *sp* and *pr*, thus keeping track of surrounding abortion and suspension conditions. The essential key to solve schizophrenia problems is to compile the surface and depth parts of statements separately, so that the occurrences of the local variables in the surface parts can be renamed according to their incarnation levels. The complete pseudo code for the compiler is given in the appendix. Its core are the two sets of functions XSurface (Figure 2) and XDepth (Figure 3), which compile the surface and depth part of a given statement type *X*. Both are initially triggered by the function StartCompile, which also subsequently combines the result of both functions for the final AIF file.

The function Surface computes the surface part with help of the following parameters:

- *L* is the list of local declarations
- $\rho$  is the substitution for renaming local names
- $go^s$  is the condition for executing the data actions of the surface
- $go^d$  is the condition for entering the depth of the statement
- *S* the statement to be compiled

The result is a tuple  $(I, \mathcal{D}^s, \mathcal{T}, \mathcal{M}^s, \mathcal{C}^s)$ , where

- *I* is the condition that holds iff the statement is instantaneous (i.e.  $inst(S)$ ),
- $\mathcal{D}^s$  is the list of guarded actions of the surface,
- $\mathcal{T}$  is the list of actions of the surface to transfer values determined by renamed local variables  $\xi$  to the corresponding variable  $x$  in the depth ( $\mathcal{T}$  is a list of pairs  $(x, (go^d, \xi))$ ),
- $\mathcal{M}^s$  is the list of module calls made in the surface (each one consists of an instance name, the module's name, the argument expressions with a possible disabling condition, as well as the surface context conditions  $go^s, go^d, I$  to connect the module call later by the linker, and
- $\mathcal{C}^s$  is the list of guarded actions for the control flow (delayed actions  $\phi \Rightarrow next(\ell) = true$  for a label  $\ell$ ).

There are no real surprises in the code for the surface, so that there is not much to explain. Only the translation of delayed actions on local variables needs more explanation: If a surface is activated and weakly preempted (i.e.  $go^s \wedge \neg go^d$  holds), then  $go^s$  is usually used as guard of actions of the data flow except for delayed actions on local variables: These delayed actions must not be enabled, since their local scope is left after the current step. Hence,  $go^d$  is used instead of  $go^s$  for delayed actions on local variables.

The function Depth computes the depth of a statement. It takes the following parameters:

- *h*: the depth of the next surface (an integer to rename local names)
- *sp*: condition to suspend the control flow
- *ab*: condition to abort the control flow
- *pr*: condition to preempt the data flow (i.e. the disjunction of the surrounding strong abort/suspend conditions)
- *S* the statement to be compiled

The result of this function is a tuple  $(L, N, A, T, \mathcal{D}^d, \mathcal{A}, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d)$  with:

- *L* is the list of locally declared variables of *S*

- *N* is the list of the module instance names
- *A* is equivalent to  $insd(S)$
- *T* is equivalent to  $term(S)$
- $\mathcal{D}^d$  is the list of guarded actions for the depth of *S*
- $\mathcal{A}$  is the list of absence reactions for the locally declared variables (a list of triples  $(lhs, cL, dv)$  where *lhs* is a lhs of a locally decl. variable, *cL* is list of the guarded cases and *dv* the default value), renamed surfaces to the corresponding local variable in the depth,
- $\mathcal{M}^s$  is the list of module calls to surfaces of other modules contained in the depth (see sequences, loops, and immediate suspension),
- $\mathcal{M}^d$  is the list of module calls to depths of other modules that are called in the depth,
- $\mathcal{C}^d$  is the list of transition equations for the depth of *S*

The compilation of the depth of a statement is done by a function call  $Depth(h, sp, ab, pr, S)$ . In addition to the statement *S*, the function call has the parameters *sp*, *ab* and *pr*, which indicate whether a surrounding suspension or abortion statement is preempting the execution. These conditions are refined each time a new context of a preemption statement is traversed. The compilation of the depth of a statement leads to subsequent calls to the depth compilation of its sub-statements, and it may additionally lead to the compilation of some surfaces of the sub-statements. This is the case for sequences, loops and the immediate weak suspension statement. The compilation of the depth must then not only provide an adequate renaming  $\rho$ , but must also provide the start signals  $go^s$  and  $go^d$  for the surface compilation.

## V. EXAMPLE

Consider the following two modules:

```

module  $M_1$ (bool ?i, bool !o) {
    if(i) {  $\ell$  : pause;  $o = i$ ; } else  $o = true$ ;
}
    
```

- surface (context:  $inst(M_1) = \neg i$ )
  - control flow:  $\{(go^d(M_1) \wedge i, next(\ell) = true)\}$
  - data flow:  $\{(go^s(M_1) \wedge \neg i, o = true)\}$
- depth (context:  $insd(M_1) = \ell$ ;  $term(M_1) = \ell$ )
  - data flow:  $\{(\ell \wedge \neg prmt(M_1), o = i)\}$

```

module  $M_2$ (bool ?i, bool !o) {
    loop {
        bool b;
        weak abort  $M_1(i, b)$ ; when(b);
         $\ell$  : pause;
         $o = b$ ;
    }
}
    
```

- surface (context:  $inst(M_2) = false$ )
  - control flow:  $\{(go^d(M_2), next(\ell) = true)\}$
- depth  $\{(insd(M_2) = insd(M_1) \vee \ell, term(M_2) = false)\}$ 
  - control flow:  $\{(\neg abrt(M_2) \wedge \ell \wedge term(M_1) \vee b^1, next(\ell) = true)\}$
  - data flow:  $\{(\ell \wedge \neg prmt(M_2), o = b)\}$

- absence  
–  $(b, \text{if } \text{go}^d(M_2) \text{ then } = b^1 \text{ else false})$

In addition, module  $M_2$  stores three references to  $M_1$  in its intermediate format: one depth and two surfaces, one with parameter  $b$  and the other one with its reincarnation as parameter,  $b_1$ .

## VI. CONCLUSIONS

In this paper, we show how modular/separate compilation for imperative synchronous languages can be achieved. 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. The reincarnation problem is handled by reincarnating the module calls with a sophisticated qualification of variable names so that the linker has not to care about local variables (hence, local variables are for the first time really treated locally in the compilation!). We developed a translation procedure for the whole language Quartz, including programs containing schizophrenia problems.

## 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. In *Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.
- [4] G. Berry. Synchronous languages for hardware and software reactive systems. In C. Delgado Kloos and E. Cerny, editors, *Conference on Computer Hardware Description Languages and Their Applications (CHDL)*, Toledo, Spain, 1997. Chapman & Hall.
- [5] 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.
- [6] G. Berry. The constructive semantics of pure Esterel. <http://www-sop.inria.fr/esterel.org/>, July 1999.
- [7] G. Berry. The Esterel v5 language primer. <http://www-sop.inria.fr/esterel.org/>, July 2000.
- [8] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [9] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison Wesley, Austin, Texas, May 1989.
- [10] 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.
- [11] 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), 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).
- [12] 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, NJ, USA, 1996. Springer.
- [13] S. Edwards. An Esterel compiler for large control-dominated systems. *IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems*, 21(2):169–183, February 2002.
- [14] S. Edwards. ESUIF: An open Esterel compiler. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 65(5), 2002. Workshop on Synchronous Languages, Applications, and Programming (SLAP).
- [15] S.A. Edwards. Compiling Esterel into sequential code. In *International Workshop on Hardware/Software Codesign (CODES)*, pages 147–151, Rome, Italy, 1999.
- [16] S.A. Edwards, V. Kapadia, and M. Halas. Compiling Esterel into static discrete-event code. In *Synchronous Languages, Applications, and Programming (SLAP)*, Barcelona, Spain, 2004.
- [17] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
- [18] 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.
- [19] L. Lamport. The temporal logic of actions. Technical Report 79, Digital Equipment Cooperation, 1991.
- [20] L. Lavagno and E. Sentovich. ECL: A specification environment for system-level design. In *Design Automation Conference (DAC)*, pages 511–516, New Orleans, Louisiana, USA, 1999. ACM.
- [21] 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.
- [22] D. Potop-Butucaru and R. de Simone. Optimizations for faster execution of Esterel programs. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 227–236, Mont Saint-Michel, France, 2003. IEEE Computer Society.
- [23] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In *Proceedings of the Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 195–208. Springer, 1991.
- [24] K. Schneider. A verified hardware synthesis for Esterel. In F.J. Rammig, editor, *Workshop on Distributed and Parallel Embedded Systems (DIPES)*, pages 205–214, Schloß Ehringerfeld, Germany, 2000. Kluwer.
- [25] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 143–156, Newcastle upon Tyne, UK, 2001. IEEE Computer Society.
- [26] K. Schneider. Proving the equivalence of microstep and macrostep semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logic (TPHOL)*, volume 2410 of *LNCS*, pages 314–331, Hampton, VA, USA, 2002. Springer.
- [27] K. Schneider. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, 2009.
- [28] K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Conference on Application of Concurrency to System Design (ACSD)*, page 78, Xi’an, China, 2008. IEEE Computer Society.
- [29] 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.
- [30] 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.
- [31] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, 2005. IEEE Computer Society.
- [32] K. Schneider, J. Brandt, and E. Vecchié. Efficient code generation from synchronous programs. In 165–174, editor, *Formal Methods and Models for Codesign (MEMOCODE)*, pages 165–174, Napa Valley, California, 2006. IEEE Computer Society.
- [33] K. Schneider, J. Brandt, and E. Vecchié. Modular compilation of synchronous programs. In *IFIP Conference on Distributed and Parallel Embedded Systems (DIPES)*, pages 75–84, Braga, Portugal, 2006. Springer.
- [34] K. Schneider and M. Wenz. A new method for compiling schizophrenic synchronous programs. In *Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 49–58, Atlanta, Georgia, USA, 2001. ACM.
- [35] T.R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996.
- [36] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 39–48, San Diego, California, USA, 2004. IEEE Computer Society.
- [37] 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 (ICCESS)*, volume 3820 of *LNCS*, pages 129–140, Xi’an, China, 2005. Springer.
- [38] Y. Zhou and E.A. Lee. A causality interface for deadlock analysis in dataflow. In S.L. Min and W. Yi, editors, *International Conference on Embedded Software (EMSOFT)*, pages 44–52, Seoul, Korea, 2006. ACM.

<pre> <b>function</b> CompileModule(module Module(args){S})   go<sup>s</sup> := NewParam();   go<sup>d</sup> := NewParam();   sp := NewParam();   ab := NewParam();   pr := NewParam();   (I, D<sup>s</sup>, T, M<sub>1</sub><sup>s</sup>, C<sup>s</sup>) := Surface({}, {}, go<sup>s</sup>, go<sup>d</sup>, S);   (L, N, A, T, D<sup>d</sup>, A, M<sub>2</sub><sup>s</sup>, M<sup>d</sup>, C<sup>d</sup>) = Depth(0, sp, ab, pr, S);   <b>return</b> (     module, args, L, (I, D<sup>s</sup>, M<sub>1</sub><sup>s</sup>, C<sup>s</sup>),     (A, T, D<sup>d</sup>, M<sub>2</sub><sup>s</sup>, M<sup>d</sup>, C<sup>d</sup>), A   )  <b>function</b> CombineABS(A, T)   A' := {};   <b>forall</b> (x, T<sub>1</sub>, d<sub>x</sub>) ∈ A   T' := T' ∪ ∪<sub>(x, ψ<sub>x</sub>, v<sub>x</sub>) ∈ T</sub> (x, ψ<sub>x</sub>, v<sub>x</sub>);   A' := A' ∪ T';   <b>return</b> (A')  <b>function</b> SurfaceRenaming(h, L, N)   ρ<sub>1</sub> := ∪<sub>(α, x) ∈ L</sub> (x ↦ x<sup>h</sup>);   ρ<sub>2</sub> := ∪<sub>x ∈ N</sub> (x ↦ x<sup>h</sup>);   <b>return</b> (ρ<sub>1</sub> ∪ ρ<sub>2</sub>)         </pre>	<pre> <b>function</b> DisableDelayedAssignment(ψ, x, G)   G<sub>1</sub> := { γ ∧ ψ ⇒ next(x) = τ   (γ ⇒ A) ∈ G ∧           A = (next(x) = τ) }   G<sub>2</sub> := { γ ⇒ next(x) = τ   (γ ⇒ A) ∈ G ∧           A ≠ (next(x) = τ) }   <b>return</b> (G<sub>1</sub> ∪ G<sub>2</sub>)  <b>function</b> DisableDelayedArgument(ψ, x, M)   M' := {};   <b>forall</b> (C, inst, Module, ⟨(p<sub>0</sub>, a<sub>0</sub>), …, (p<sub>n</sub>, a<sub>n</sub>)⟩) ∈ M   <b>for</b> i ∈ {0, …, n}     <b>if</b> p<sub>0</sub> = x <b>then</b> a'<sub>0</sub> := a<sub>0</sub> ∧ ψ <b>else</b> a'<sub>0</sub> := a<sub>0</sub>;   M' := M' ∪ (C, inst, Module, ⟨(p<sub>0</sub>, a'<sub>0</sub>), …, (p<sub>n</sub>, a'<sub>n</sub>)⟩);   <b>return</b> (M')         </pre>
--	--

Fig. 1. Compilation Algorithm (1/3): Start and Auxiliary Functions

<pre> <b>function</b> Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S)   <b>case</b> S <b>of</b>   nothing :     <b>return</b> (true, {}, {}, {}, {});   ℓ : pause :     <b>return</b> (false, {}, {}, {}, {go<sup>d</sup> ⇒ next(ℓ) = true})   x = τ :     <b>return</b> (true, {(go<sup>s</sup>, ρ(x = τ))}, {}, {}, {})   next(x) = τ :     <b>if</b> x ∈ L <b>then</b> γ := go<sup>d</sup> <b>else</b> γ := go<sup>s</sup>     <b>return</b> (true, {γ ⇒ ρ(next(x) = τ)}, {}, {}, {})   <b>if</b> (σ) S<sub>1</sub> <b>else</b> S<sub>2</sub> :     <b>return</b> ConditionalSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, σ, S<sub>1</sub>, S<sub>2</sub>);   S<sub>1</sub>; S<sub>2</sub> :     <b>return</b> SequenceSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S<sub>1</sub>, S<sub>2</sub>);   S<sub>1</sub>    S<sub>2</sub> :     <b>return</b> ParallelSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S<sub>1</sub>, S<sub>2</sub>);   <b>do</b> S <b>while</b> (σ) :     <b>return</b> Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S)   <b>abort</b> S <b>when</b> (σ) :     <b>return</b> Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S)   <b>weak abort</b> S <b>when</b> (σ) :     <b>return</b> Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S)   <b>abort</b> S <b>when immediate</b>(σ) :     <b>return</b> AbortImmSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, σ, S, false);   <b>weak abort</b> S <b>when immediate</b>(σ) :     <b>return</b> AbortImmSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, σ, S, true);   <b>suspend</b> S <b>when</b> (σ) :     <b>return</b> Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S)   <b>weak suspend</b> S <b>when</b> (σ) :     <b>return</b> Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S)   ℓ : <b>suspend</b> S <b>when immediate</b>(σ) :     <b>return</b> SuspendImmSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, ℓ, false, σ, S, false);   ℓ : <b>weak suspend</b> S <b>when immediate</b>(σ) :     <b>return</b> SuspendImmSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, ℓ, true, σ, S, true);   {α x; S} :     (I, D<sup>s</sup>, T<sub>1</sub>, M<sup>s</sup>, C<sup>s</sup>) := Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S);     <b>if</b> storage(α) = memorized       <b>then</b> T<sub>2</sub> = {(x, go<sup>d</sup>, pre(x))} <b>else</b> T<sub>2</sub> = {};       M<sup>s</sup> = DisableDelayedArgument(M<sub>1</sub><sup>s</sup>, x, ¬go<sup>d</sup>);       <b>return</b> (I, D<sup>s</sup>, T<sub>1</sub> ∪ T<sub>2</sub>, M<sup>s</sup>, C<sup>s</sup>);   <b>inst</b> : name(argL) :     I := NewParam();     M<sup>s</sup> := {(go<sup>s</sup>, go<sup>d</sup>, I), inst, name, argL};     <b>return</b> (I, {}, {}, M<sup>s</sup>, {});         </pre>	<pre> <b>function</b> ConditionalSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, σ, S<sub>1</sub>, S<sub>2</sub>)   go<sub>1</sub><sup>s</sup> := NewDef(go<sup>s</sup> ∧ ρ(σ));   go<sub>1</sub><sup>d</sup> := NewDef(go<sup>d</sup> ∧ ρ(σ));   go<sub>2</sub><sup>s</sup> := NewDef(go<sup>s</sup> ∧ ¬ρ(σ));   go<sub>2</sub><sup>d</sup> := NewDef(go<sup>d</sup> ∧ ¬ρ(σ));   (I<sub>1</sub>, D<sub>1</sub><sup>s</sup>, T<sub>1</sub>, M<sub>1</sub><sup>s</sup>, C<sub>1</sub><sup>s</sup>) := Surface(L, ρ, go<sub>1</sub><sup>s</sup>, go<sub>1</sub><sup>d</sup>, S<sub>1</sub>);   (I<sub>2</sub>, D<sub>2</sub><sup>s</sup>, T<sub>2</sub>, M<sub>2</sub><sup>s</sup>, C<sub>2</sub><sup>s</sup>) := Surface(L, ρ, go<sub>2</sub><sup>s</sup>, go<sub>2</sub><sup>d</sup>, S<sub>2</sub>);   I := NewDef(I<sub>1</sub> ∧ ρ(σ) ∨ I<sub>2</sub> ∧ ¬ρ(σ) ∨ I<sub>1</sub> ∧ I<sub>2</sub>);   <b>return</b> (I, D<sub>1</sub><sup>s</sup> ∪ D<sub>2</sub><sup>s</sup>, T<sub>1</sub> ∪ T<sub>2</sub>, M<sub>1</sub><sup>s</sup> ∪ M<sub>2</sub><sup>s</sup>, C<sub>1</sub><sup>s</sup> ∪ C<sub>2</sub><sup>s</sup>);  <b>function</b> ParallelSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S<sub>1</sub>, S<sub>2</sub>)   (I<sub>1</sub>, D<sub>1</sub><sup>s</sup>, T<sub>1</sub>, M<sub>1</sub><sup>s</sup>, C<sub>1</sub><sup>s</sup>) := Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S<sub>1</sub>);   (I<sub>2</sub>, D<sub>2</sub><sup>s</sup>, T<sub>2</sub>, M<sub>2</sub><sup>s</sup>, C<sub>2</sub><sup>s</sup>) := Surface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S<sub>2</sub>);   I := NewDef(I<sub>1</sub> ∧ I<sub>2</sub>);   <b>return</b> (I, D<sub>1</sub><sup>s</sup> ∪ D<sub>2</sub><sup>s</sup>, T<sub>1</sub> ∪ T<sub>2</sub>, M<sub>1</sub><sup>s</sup> ∪ M<sub>2</sub><sup>s</sup>, C<sub>1</sub><sup>s</sup> ∪ C<sub>2</sub><sup>s</sup>);  <b>function</b> SequenceSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, S<sub>1</sub>, S<sub>2</sub>)   (I<sub>1</sub>, D<sub>1</sub><sup>s</sup>, T<sub>1</sub>, M<sub>1</sub><sup>s</sup>, C<sub>1</sub><sup>s</sup>) := Surface(L, ρ, go<sub>1</sub><sup>s</sup>, go<sub>1</sub><sup>d</sup>, S<sub>1</sub>);   go<sub>2</sub><sup>s</sup> := NewDef(go<sup>s</sup> ∧ I<sub>1</sub>);   go<sub>2</sub><sup>d</sup> := NewDef(go<sup>d</sup> ∧ I<sub>1</sub>);   (I<sub>2</sub>, D<sub>2</sub><sup>s</sup>, T<sub>2</sub>, M<sub>2</sub><sup>s</sup>, C<sub>2</sub><sup>s</sup>) := Surface(L, ρ, go<sub>2</sub><sup>s</sup>, go<sub>2</sub><sup>d</sup>, S<sub>2</sub>);   I := NewDef(I<sub>1</sub> ∧ I<sub>2</sub>);   <b>return</b> (I, D<sub>1</sub><sup>s</sup> ∪ D<sub>2</sub><sup>s</sup>, T<sub>1</sub> ∪ T<sub>2</sub>, M<sub>1</sub><sup>s</sup> ∪ M<sub>2</sub><sup>s</sup>, C<sub>1</sub><sup>s</sup> ∪ C<sub>2</sub><sup>s</sup>);  <b>function</b> AbortImmSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, σ, S, wk)   <b>if</b> wk <b>then</b>     go<sub>1</sub><sup>s</sup> := go<sup>s</sup>;   <b>else</b>     go<sub>1</sub><sup>s</sup> := NewDef(go<sup>s</sup> ∧ ¬ρ(σ));     go<sub>1</sub><sup>d</sup> := NewDef(go<sup>d</sup> ∧ ¬ρ(σ));     (I<sub>1</sub>, D<sup>s</sup>, T, M<sup>s</sup>, C<sup>s</sup>) := Surface(L, ρ, go<sub>1</sub><sup>s</sup>, go<sub>1</sub><sup>d</sup>, S);     I := NewDef(I<sub>1</sub> ∨ ρ(σ));     <b>return</b> (I, D<sup>s</sup>, T, M<sup>s</sup>, C<sup>s</sup>);  <b>function</b> SuspendImmSurface(L, ρ, go<sup>s</sup>, go<sup>d</sup>, ℓ, σ, S)   c := {go<sup>d</sup> ∧ ρ(σ) ⇒ next(ℓ) = true};   <b>if</b> wk <b>then</b>     go<sub>1</sub><sup>s</sup> := go<sup>s</sup>;   <b>else</b>     go<sub>1</sub><sup>s</sup> := NewDef(go<sup>s</sup> ∧ ¬ρ(σ));     go<sub>1</sub><sup>d</sup> := NewDef(go<sup>d</sup> ∧ ¬ρ(σ));     (I<sub>1</sub>, D<sup>s</sup>, T, M<sup>s</sup>, C<sup>s</sup>) := Surface(L, ρ, go<sub>1</sub><sup>s</sup>, go<sub>1</sub><sup>d</sup>, S);     I := NewDef(I<sub>1</sub> ∧ ¬ρ(σ));     <b>return</b> (I, D<sup>s</sup>, T, M<sup>s</sup>, c ∪ C<sup>s</sup>);         </pre>
---	---

Fig. 2. Compilation Algorithm (2/3): Surface Computation

```

function Depth( $\bar{h}$ , sp, ab, pr, S)
  case S of
    nothing :
      return ( $\{\}, \{\}, \text{false}, \text{false}, \{\}, \{\}, \{\}, \{\}, \{\}$ )
     $\ell$  : pause :
       $\text{go}^d := \text{NewDef}(\text{sp} \wedge \ell)$ ;
       $\mathcal{C}^d := \{\text{go}^d \Rightarrow \text{next}(\ell) = \text{true}\}$ ;
      return ( $\{\}, \{\}, \ell, \ell, \{\}, \{\}, \{\}, \{\}, \mathcal{C}^d$ )
     $x = \tau, \text{next}(x) = \tau$  :
      return ( $\{\}, \{\}, \text{false}, \text{false}, \{\}, \{\}, \{\}, \{\}, \{\}$ )
    if( $\sigma$ )  $S_1$  else  $S_2$  :
      return ConditionalDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ ,  $S_1$ ,  $S_2$ )
     $S_1; S_2$  :
      return SequenceDepth( $\bar{h}$ , sp, ab, pr,  $S_1$ ,  $S_2$ )
     $S_1 \parallel S_2$  :
      return ParallelDepth( $\bar{h}$ , sp, ab, pr,  $S_1$ ,  $S_2$ )
    do S while( $\sigma$ ) :
      return DoWhileDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S)
    about S when( $\sigma$ ) :
      return AbortDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, false)
    about S when immediate( $\sigma$ ) :
      return AbortDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, false)
    weak about S when( $\sigma$ ) :
      return AbortDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, true)
    weak about S when immediate( $\sigma$ ) :
      return AbortDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, true)
    suspend S when( $\sigma$ ) :
      return SuspendDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, false)
    weak suspend S when( $\sigma$ ) :
      return SuspendDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, true)
     $\ell$  : suspend S when immediate( $\sigma$ ) :
      return SuspendImmDepth( $\bar{h}$ , sp, ab, pr,  $\ell$ ,  $\sigma$ , S, false)
     $\ell$  : weak suspend S when immediate( $\sigma$ ) :
      return SuspendImmDepth( $\bar{h}$ , sp, ab, pr,  $\ell$ ,  $\sigma$ , S, true)
     $\{\alpha x; S\}$  :
      ( $L, N, A, T, \mathcal{D}_1^d, \mathcal{A}_1, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ ) =
        Depth( $\bar{h}$ , sp, ab, pr, S);
       $\mathcal{D}^d = \text{DisableDelayedLocals}(\mathcal{D}_1^d, x, T)$ ;
       $\mathcal{M}^d = \text{DisableDelayedArguments}(\mathcal{M}_1^d, T)$ ;
       $\mathcal{A}_2 = \{x, \{\}, \text{Default}(\alpha)\}$ ;
      return ( $L \cup \{(\alpha, x)\}, N, A, T, \mathcal{D}^d,$ 
         $\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ )
    inst : name(argL) :
       $A := \text{NewParam}()$ ;
       $T := \text{NewParam}()$ ;
       $\mathcal{M}^d = \{(\text{sp}, \text{ab}, \text{pr}, T), \text{inst}, \text{name}, \text{argL}\}$ ;
      return ( $\{\}, \{\}, A, T, \{\}, \{\}, \{\}, \mathcal{M}^d, \{\}$ )

function ConditionalDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ ,  $S_1$ ,  $S_2$ )
  ( $L_1, N_1, A_1, T_1, \mathcal{D}_1^d,$ 
   $\mathcal{A}_1, \mathcal{M}_1^s, \mathcal{M}_1^d, \mathcal{C}_1^d$ ) := Depth( $\bar{h}$ , sp, ab, pr,  $S_1$ );
  ( $L_2, N_2, A_2, T_2, \mathcal{D}_2^d,$ 
   $\mathcal{A}_2, \mathcal{M}_2^s, \mathcal{M}_2^d, \mathcal{C}_2^d$ ) := Depth( $\bar{h}$ , sp, ab, pr,  $S_2$ );
   $A := \text{NewDef}(A_1 \vee A_2)$ ;
   $T := \text{NewDef}(T_1 \vee T_2)$ ;
  return ( $L_1 \cup L_2, N_1 \cup N_2, A, T, \mathcal{D}_1^d \cup \mathcal{D}_2^d,$ 
   $\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{M}_1^s \cup \mathcal{M}_2^s, \mathcal{M}_1^d \cup \mathcal{M}_2^d, \mathcal{C}_1^d \cup \mathcal{C}_2^d$ )

function ParallelDepth( $\bar{h}$ , sp, ab, pr,  $S_1$ ,  $S_2$ )
  ( $L_1, N_1, A_1, T_1, \mathcal{D}_1^d,$ 
   $\mathcal{A}_1, \mathcal{M}_1^s, \mathcal{M}_1^d, \mathcal{C}_1^d$ ) := Depth( $\bar{h}$ , sp, ab, pr,  $S_1$ );
  ( $L_2, N_2, A_2, T_2, \mathcal{D}_2^d,$ 
   $\mathcal{A}_2, \mathcal{M}_2^s, \mathcal{M}_2^d, \mathcal{C}_2^d$ ) := Depth( $\bar{h}$ , sp, ab, pr,  $S_2$ );
   $A := \text{NewDef}(A_1 \vee A_2)$ ;
   $T := \text{NewDef}(T_1 \wedge \neg A_2 \vee T_2 \wedge \neg A_1 \vee T_1 \wedge T_2)$ ;
  return ( $L_1 \cup L_2, N_1 \cup N_2, A, T, \mathcal{D}_1^d \cup \mathcal{D}_2^d,$ 
   $\mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{M}_1^s \cup \mathcal{M}_2^s, \mathcal{M}_1^d \cup \mathcal{M}_2^d, \mathcal{C}_1^d \cup \mathcal{C}_2^d$ )

function SequenceDepth( $\bar{h}$ , sp, ab, pr,  $S_1$ ,  $S_2$ )
  ( $L_1, N_1, A_1, T_1, \mathcal{D}_1^d, \mathcal{A}_1, \mathcal{M}_1^s, \mathcal{M}_1^d, \mathcal{C}_1^d$ ) := Depth( $\bar{h}$ , sp, ab, pr,  $S_1$ )
   $\text{go}_1^s := \text{NewDef}(T_1 \wedge \neg \text{pr})$ ;
   $\text{go}_2^d := \text{NewDef}(\text{go}_1^s \wedge \neg(\text{sp} \vee \text{ab}))$ ;
  ( $I_2, \mathcal{D}_2^s, T_2, \mathcal{M}_2^s, \mathcal{C}_2^s$ ) := Surface( $\{\}, \{\}, \text{go}_1^s, \text{go}_2^d, S_2$ )
  ( $L_2, N_2, A_2, T_2, \mathcal{D}_2^d, \mathcal{A}_2, \mathcal{M}_2^s, \mathcal{M}_2^d, \mathcal{C}_2^d$ ) := Depth( $\bar{h}$ , sp, ab, pr,  $S_2$ )
   $A = \text{NewDef}(A_1 \vee A_2)$ ;
   $T = \text{NewDef}(T_1 \wedge I_2 \vee T_2)$ ;
  return ( $L_1 \cup L_2, N_1 \cup N_2, A, T, \mathcal{D}_1^d \cup \mathcal{D}_2^d \cup \mathcal{D}_2^s, \mathcal{A}_1 \cup \mathcal{A}_2,$ 
   $\mathcal{M}_1^s \cup \mathcal{M}_2^s \cup \mathcal{M}_2^s, \mathcal{M}_1^d \cup \mathcal{M}_2^d, \mathcal{C}_1^d \cup \mathcal{C}_2^d$ )

function DoWhileDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S)
  ( $L, N, A, T_1, \mathcal{D}^d, \mathcal{A}_1, \mathcal{M}_1^s, \mathcal{M}^d, \mathcal{C}^d$ ) = Depth( $\bar{h} + 1$ , sp, ab, pr, S)
  // compile the surface for re-entering the loop body
   $\text{go}^s := \text{NewDef}(T_1 \wedge \sigma \wedge \neg \text{pr})$ ;
   $\text{go}^d := \text{NewDef}(\text{go}^s \wedge \neg(\text{ab} \vee \text{sp}))$ ;
   $\rho := \text{MkSurfaceRenaming}(\bar{h}, L, N)$ ;
  ( $I, \mathcal{D}^s, T, \mathcal{M}_2^s, \mathcal{C}^s$ ) = Surface( $L, \rho, \text{go}^s, \text{go}^d, S$ );
   $\mathcal{A} := \text{CombineABS}(\mathcal{A}_1, T)$ ;
   $T := \text{NewDef}(T \wedge \neg \sigma)$ ;
  return ( $L, N, A, T, \mathcal{D}^s \cup \mathcal{D}^d, \mathcal{A}, \mathcal{M}_1^s \cup \mathcal{M}_2^s, \mathcal{M}^d, \mathcal{C}^s \cup \mathcal{C}^d$ )

function AbortDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, wk)
   $\text{ab}_1 := \text{NewDef}(\text{ab} \vee \sigma)$ ;
  if wk then
    ( $L, N, A, T_1, \mathcal{D}^d, \mathcal{A}, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ ) = Depth( $\bar{h}$ , sp,  $\text{ab}_1$ , pr, S)
  else
     $\text{pr}_1 := \text{NewDef}(\text{pr} \vee \sigma)$ ;
    ( $L, N, A, T_1, \mathcal{D}^d, \mathcal{A}, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ ) = Depth( $\bar{h}$ , sp,  $\text{ab}_1$ ,  $\text{pr}_1$ , S)
   $T := \text{NewDef}(T_1 \vee A \wedge \text{ab})$ ;
  return ( $L, N, A, T, \mathcal{D}^d, \mathcal{A}, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ )

function SuspendDepth( $\bar{h}$ , sp, ab, pr,  $\sigma$ , S, wk)
   $\text{sp}_1 := \text{NewDef}(\text{sp} \vee \sigma \wedge \neg \text{ab})$ ;
  if wk then
    ( $L, N, A, T_1, \mathcal{D}^d, \mathcal{A}, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ ) = Depth( $\bar{h}$ ,  $\text{sp}_1$ , ab, pr, S)
  else
     $\text{pr}_1 := \text{NewDef}(\text{pr} \vee \sigma)$ ;
    ( $L, N, A, T_1, \mathcal{D}^d, \mathcal{A}, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ ) = Depth( $\bar{h}$ ,  $\text{sp}_1$ ,  $\text{ab}$ ,  $\text{pr}_1$ , S)
   $T := \text{NewDef}(T_1 \wedge \neg \text{sp}_1)$ ;
  return ( $L, N, A, T, \mathcal{D}^d, \mathcal{A}, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}^d$ )

function SuspendImmDepth( $\bar{h}$ , sp, ab, pr,  $\ell$ ,  $\sigma$ , S, wk)
   $\text{sp}_1 := \text{NewDef}(\text{sp} \vee \sigma \wedge \neg \text{ab})$ ;
   $\mathcal{C}_1^d := \{\ell \wedge \text{sp}_1 \Rightarrow \text{next}(\ell) = \text{true}\}$ ;
  if wk then
     $\text{go}_1^s := \text{NewDef}(\ell_1)$ ;
     $\text{go}_1^d := \text{NewDef}(\text{go}_1^s \wedge \neg(\text{sp}_1 \vee \text{ab}))$ ;
    ( $I_2, \mathcal{D}_2^s, T, \mathcal{M}_2^s, \mathcal{C}_2^s$ ) = Surface( $\{\}, \{\}, \text{go}_1^s, \text{go}_1^d, S$ )
    ( $L, N, A_2, T_2, \mathcal{D}_2^d, \mathcal{A}_2, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}_2^d$ ) = Depth( $\bar{h}$ ,  $\text{sp}_1$ , ab, pr, S);
  else
     $\text{pr}_1 := \text{NewDef}(\text{pr} \vee \sigma)$ ;
     $\text{go}_1^s := \text{NewDef}(\ell_1 \wedge \neg \text{pr}_1)$ ;
     $\text{go}_1^d := \text{NewDef}(\text{go}_1^s \wedge \neg(\text{sp}_1 \vee \text{ab}))$ ;
    ( $I_2, \mathcal{D}_2^s, T, \mathcal{M}_2^s, \mathcal{C}_2^s$ ) = Surface( $\{\}, \{\}, \text{go}_1^s, \text{go}_1^d, S$ );
    ( $L, N, A_2, T_2, \mathcal{D}_2^d, \mathcal{A}_2, \mathcal{M}^s, \mathcal{M}^d, \mathcal{C}_2^d$ ) = Depth( $\bar{h}$ ,  $\text{sp}_1$ , ab,  $\text{pr}_1$ , S);
   $A_1 := \text{NewDef}(\ell \vee A_2)$ ;
   $T_1 := \text{NewDef}(\ell \wedge I_2)$ ;
   $T := \text{NewDef}((T_1 \vee T_2) \wedge \neg \sigma)$ ;
  return ( $L, N, A, T, \mathcal{D}_2^s \cup \mathcal{D}_2^d, \mathcal{A},$ 
   $\mathcal{M}^s \cup \mathcal{M}_2^s, \mathcal{M}^d, \mathcal{C}_1^d \cup \mathcal{C}_2^s \cup \mathcal{C}_2^d$ )

```

Fig. 3. Compilation Algorithm (3/3): Depth Computation