A Verified Compiler for Synchronous Programs with Local Declarations

Klaus Schneider, Jens Brandt, and Tobias Schuele

University of Kaiserslautern
Department of Computer Science
Reactive Systems Group
P.O. Box 3049, 67653 Kaiserslautern, Germany
http://rsg.informatik.uni-kl.de
Email: firstname.lastname@informatik.uni-kl.de

Abstract

We describe the translation of Esterel-like programs with delayed actions to equivalent equation systems. Potential schizophrenia problems arising from local declarations are solved by (1) generating copies of the surface of the statement and (2) renaming the local variables in one of the copied surfaces generated a loop. The translation runs in quadratic time and has been formally verified with the HOL theorem prover.

1 Introduction

Synchronous languages [1] like Esterel [2,4] and its variants [12,16] offer a convenient programming paradigm for the design of reactive real-time systems. Several success stories have been reported [8] from safety-critical applications like avionic and automotive industries, transportation, and many others. The clear formal semantics of these languages allows us to apply formal methods not only to verify particular programs, but also to reason about the semantics itself, e.g., to verify program transformations [18,17].

The common paradigm of these languages is the *perfect synchrony* [1], which means that most of the statements are executed as *micro steps* in zero time. Consumption of time is explicitly programmed by partitioning the micro steps into 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 the micro steps of a macro step are executed at the same point of time, their ordering within the macro step is irrelevant (provided that dependency cycles have been eliminated). Therefore, values of variables are determined with respect to macro steps instead of micro steps.

The abstraction to macro steps simplifies the semantics of synchronous languages and yields a clear programming model for multithreaded programs and

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

hardware circuits. However, this abstraction is not for free: Causality cycles and schizophrenic statements are the two major problems that must be solved by the compilers. *Causality cycles* arise when the condition for executing an action is influenced by this action. Algorithms for causality analysis, that check if such cyclic dependencies have unique solutions, are related with the analysis of combinational feedback loops of hardware circuits [13,11,6,19,5,3]. Usually, causality analysis is performed as a second step after the compilation to an equation system.

Schizophrenia problems [14,3,18] are usually solved before causality analysis. In general, a statement is schizophrenic if some of its micro steps are executed more than once in a macro step. This may happen *only if the statement belongs to a loop body* that is left and (re)entered at the same time (in the same macro step). If the scope of a local declaration is thereby left and (re)entered, then the compiler must carefully distinguish between different incarnations of local variables that exist at the same time. To this end, the compiler has to generate copies (incarnations) of the locally declared variables. In general, several copies may be necessary since nesting of abort and loop statements may enforce several executions of the local declaration. Referring to the right incarnations of local variables poses a difficult problem for the compilers [3], since all variables must have a uniquely determined value at each point of time.

In this paper, we present a new algorithm to translate synchronous programs of the Esterel-family into a simple intermediate format: The control flow is given as an equation system [9,3,15] and the data flow is given as a set of guarded commands [16]. Schizophrenia problems are solved by computing control and data flow representations *separately for the surface and the depth of a program* [3,16,18]. Intuitively, the surface consists of the micro steps that are executed when the program is started, i.e., when the control enters the program. The depth contains the micro steps that are executed when the program resumes execution after a macro step, i.e., when the control is already inside the program. Overlapping of surface and depth parts of local declarations can lead to schizophrenia problems. Hence, computing control and data flow separately for the surface and depth allows us to generate copies of the overlapping parts (at least of the entire surface). Moreover, we can rename the local variables in the surfaces that are generated by loop bodies to cure schizophrenia problems.

We have embedded [16] our Esterel variant Quartz in the HOL theorem prover [10]. This embedding allows us to reason not only about particular Quartz programs, but also about the semantics of Quartz. In previous work, we have already proved the correctness of the synthesis of equation systems [15] and the equivalence to SOS rules [17]. The latter enables us to reason about micro steps, which is necessary to prove the correctness of the translation presented in this paper.

The paper is organized as follows: in the next section, we briefly describe differences between Esterel and Quartz [15,16,17,18]. In particular, we consider schizophrenia problems that occur in Quartz programs in Section 2.2. In Section 3, we consider previous solutions to solve schizophrenic statements, and finally present our translation in Section 4.

2 Schizophrenia Problems in Quartz

2.1 Syntax and Semantics of Quartz

Quartz [15,16,17] is a variant of Esterel [2,4,8] that extends classic Esterel by delayed assignments and emissions, asynchronous concurrency, nondeterministic choice, and inline assertions. The basic statements of Quartz are given below:

Definition 2.1 [Basic Statements of Quartz] *The set of basic statements of Quartz is the smallest set that satisfies the following rules, provided that S, S_1, and S_2 are also basic statements of Quartz, ℓ is a location variable, x is an event variable, y is a state variable, σ is a Boolean expression, and α a type:*

- **nothing** *(empty statement)*
- **emit** x *and* **emit next**(x) *(immediate/delayed emission)*
- $y := \tau$ *and* **next**$(y) := \tau$ *(immediate/delayed assignment)*
- ℓ : **pause** *(consumption of time)*
- **if** σ **then** S_1 **else** S_2 **end** *(conditional)*
- $S_1; S_2$ *(sequential composition)*
- $S_1 \parallel S_2$ *(synchronous concurrency)*
- $S_1 \mid\mid\mid S_2$ *(asynchronous concurrency)*
- **choose** S_1 [] S_2 **end** *(nondeterministic choice)*
- **do** S **while** σ *(iteration)*
- **suspend** S **when** σ *(suspension)*
- **weak suspend** S **when** σ *(weak suspension)*
- **abort** S **when** σ *(abortion)*
- **weak abort** S **when** σ *(weak abortion)*
- **local** x **in** S **end** *(local event variable)*
- **local** $y : \alpha$ **in** S **end** *(local state variable)*
- **now** σ *(instantaneous assertion)*
- **during** S **holds** σ *(invariant assertion)*

In contrast to Esterel [1], Quartz distinguishes between two kinds of variables, namely *event variables* and *state variables*, which are manipulated by emissions and assignments, respectively. State variables y are 'sticky', i.e., they store the current value until an assignment changes it. Executing a delayed assignment **next**$(y) := \tau$ means to evaluate τ in the current macro step (environment) and to assign the obtained value to y in the following macro step. Immediate assignments update y in the current macro step and are therefore rather equations than assignments.

Event variables have Boolean values, i.e., they can be either 1 or 0. An event variable x is 1 at a point of time if and only if either an immediate emission **emit** x

[1] Event variables of Quartz are called pure signals in Esterel, and state variables of Quartz resemble valued Esterel signals without a status (however, while **emit** $x(2)$; **emit** $x(2)$ is a problem in Esterel, $x := 2; x := 2$ is no problem in Quartz). There are no variables in the sense of Esterel's variables (they can be easily eliminated with local (Quartz) variables with delayed assignments).

is executed in the current macro step or a delayed emission **emit next**(x) has been executed in the previous macro step. Hence, event variables do not store their value (unless this is explicitely programmed with delayed emissions).

In the following, assignments and emissions are called *actions* which can be delayed or immediate. Delayed actions are now also available in Esterel (v7) [8]. In particular, they are useful to describe hardware circuits. The additional **pre** operator of Esterel for accessing previous values is, however, not used in Quartz.

Nondeterministic choice and asynchronous concurrency (which also introduces nondeterminism) is implemented by new inputs that are not observable [16]. Immediate assertions **now** σ require that σ currently holds, and **during** S **holds** σ requires that σ holds whenever the control is inside S [16]. The semantics of the other statements is essentially the same as in Esterel. Due to lack of space, we do not describe their semantics in detail, and refer instead to [16,15,17] and to the Esterel primer [4], which is an excellent introduction to synchronous programming.

In general, a statement S may be started at a certain point of time t_1 and may terminate at time $t_2 \geq t_1$, but it may also never terminate. If S immediately terminates when it is started ($t_2 = t_1$), it is called *instantaneous*, otherwise the control flow *enters* S, and will resume the execution from somewhere *inside* S at the next point of time. Whether a statement is instantaneous or not depends on the input variables.

There is only one basic statement where the control can rest for the next macro step, namely the **pause** statement. For this reason, we endow **pause** statements with unique Boolean valued *location variables* ℓ that are true iff the control is currently at location ℓ : **pause**. Using these location variables, the control flow of statements S is defined by the control flow predicates $\mathsf{in}\,(S)$, $\mathsf{inst}\,(S)$, $\mathsf{enter}\,(S)$, $\mathsf{term}\,(S)$, and $\mathsf{move}\,(S)$, and the data flow of S is defined by the set of guarded commands $\mathsf{guardcmd}\,(\varphi, S)$ [16]:

$\mathsf{in}\,(S)$ is the disjunction of the **pause** labels occurring in S. Therefore, $\mathsf{in}\,(S)$ holds at some point of time iff the control flow is currently at some location inside S.

$\mathsf{inst}\,(S)$ holds iff the control flow can not stay in S when S would now be started. This means that the execution of S would be instantaneous at this point of time.

$\mathsf{enter}\,(S)$ describes where the control flow will be at the next point of time when S would now be started. Clearly, $\mathsf{inst}\,(S) \rightarrow \neg\mathsf{enter}\,(S)$ holds.

$\mathsf{term}\,(S)$ describes all conditions where the control flow is currently somewhere inside S (hence, $\mathsf{term}\,(S) \rightarrow \mathsf{in}\,(S)$ holds) and wants to leave S. 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.

$\mathsf{move}\,(S)$ describes all internal moves, i.e., all possible transitions from somewhere inside S to another location inside S without temporarily leaving S.

$\mathsf{guardcmd}\,(\varphi, S)$ is a set of pairs of the form $(\gamma, \mathcal{C})$, where $\mathcal{C}$ is an action or an immediate assertion **now** σ. The meaning of $(\gamma, \mathcal{C})$ is that $\mathcal{C}$ is immediately executed whenever its guard γ holds.

The above control flow predicates as well as the guarded commands can be defined by primitive recursion [16,17] over the statement. The definition of these predicates only requires Boolean operators and the temporal next operator. Given a statement S and a start location st (often called the boot register), the transition relation of the control flow $\mathcal{R}_{\mathsf{cf}}(st, S)$ is defined as follows:

$$\mathcal{R}_{\mathsf{cf}}(st, S) :\equiv \left(\begin{array}{l} (\neg\mathsf{in}\,(S) \vee \mathsf{term}\,(S)) \wedge st \wedge \mathsf{inst}\,(S) \wedge \neg\mathsf{next}(\mathsf{in}\,(S)) \vee \\ (\neg\mathsf{in}\,(S) \vee \mathsf{term}\,(S)) \wedge st \wedge \mathsf{enter}\,(S) \vee \\ (\neg\mathsf{in}\,(S) \vee \mathsf{term}\,(S)) \wedge \neg st \wedge \neg\mathsf{next}(\mathsf{in}\,(S)) \vee \\ \mathsf{move}\,(S) \end{array} \right) \wedge \neg\mathsf{next}(st)$$

The transition relation describes the behavior for instantaneous execution, entering the statement, terminating the execution, and moving the control inside the statement. The initial condition $\mathcal{I}_{\mathsf{cf}}(st, S)$ is simply defined as $\mathcal{I}_{\mathsf{cf}}(st, S) :\equiv \neg\mathsf{in}\,(S) \wedge st$.

Besides the control flow, the guarded commands have to be computed for constructing an initial condition and a transition relation for the data flow. Details on the computation are given in [16,18] and in the appendix.

2.2 *Schizophrenia Problems in* Quartz

It is well-known in the synchronous programming language community that subtle problems may arise when local declarations are nested within loop statements. The problem is hereby that the local declaration can be left and (re)entered at the same macro step. The micro steps of such a macro step must 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. Local declarations that yield different incarnations of a local variable at the same point of time are called *schizophrenic* ([3], Chapter 12).

As an example, consider the program given on the left hand side of Figure 1. The right hand side of Figure 1 shows a corresponding control-data-flow graph. The circle nodes of this graph are control flow states that are labeled with those location variables that are currently active (including the start location st). Besides these control flow states, there are two other kinds of nodes: boxes with shadowed frames contain actions that are executed when an arc towards this node is traversed. The remaining boxes represent conditions that influence the following computation, i.e., conditional branches. The outgoing arcs of such a conditional node correspond to the ‘then’ and ‘else’ branch of the condition. For example, if the program is executed from state $\{\ell\}$ and we have $\neg k \wedge j \wedge \neg i$, then we execute the two action boxes beneath control state $\{\ell\}$ 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 $\{\ell\}$ to itself. The first one 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,

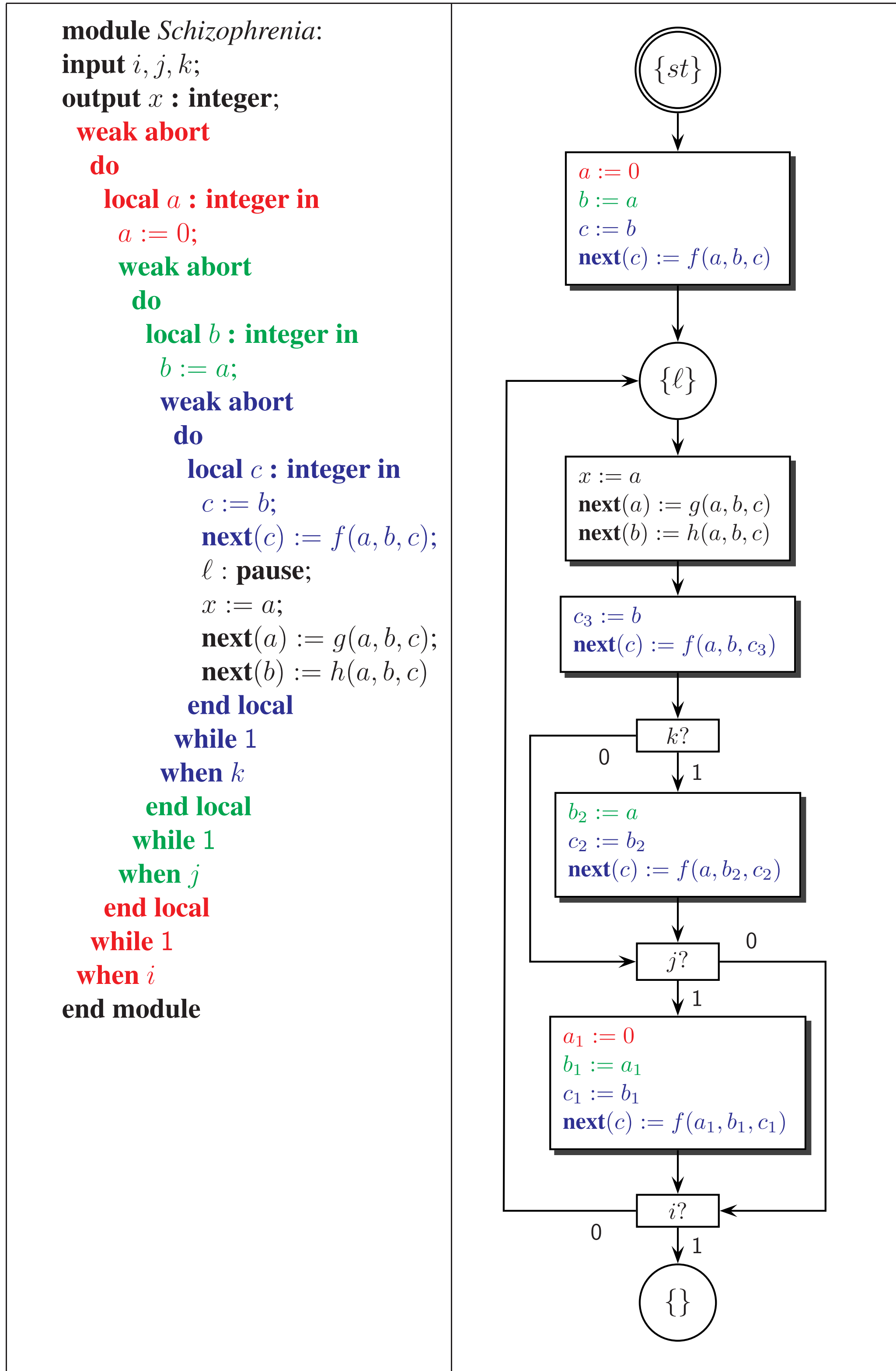

Figure 1. Local declarations with delayed actions.

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.

For software generation, one could implement the incarnations simply by shadowing the incarnations of the old scope. However, this is not possible for hardware circuit generation, since in a synchronous hardware circuit every wire has exactly one value per clock cycle. Therefore, we have to generate several copies of the locally declared variables according to the number of the possible '(re)enterings'.

Delayed actions add further difficulties to the reincarnation of locally declared variables: If a delayed action that changes the value of the locally declared variable is executed at termination time of the local declaration, then we have to disable its execution (at least when the local declaration is (re)entered at the same time). This must be done even 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. If we (re)enter the local declaration at the same point of time, we must not transfer the delayed value to the depth of the new scope.

We must also disable delayed actions on local variables in those surfaces of local declarations that do not directly proceed to their depth. Note that the surface of a local declaration can be executed more than once (see Figure 1), but at most one of these surface instances can proceed to the depth without leaving the scope. Only delayed actions of this instance of the surface are executed. For example, in Figure 1, at most one of the actions $\textbf{next}(c) := f(a, b, c)$, $\textbf{next}(c) := f(a, b, c_3)$, $\textbf{next}(c) := f(a, b_2, c_2)$, and $\textbf{next}(c) := f(a_1, b_1, c_1)$ must be executed.

Finally, note that we have to *rename all local variables*, and not only the outermost ones: Abortion statements can terminate the statement from every location, and suitable conditions for entering the statement could lead to reincarnations. Hence, it may be the case that surfaces of the nested local declarations overlap.

3 Previous Solutions to Cure Schizophrenia

Several solutions have already been proposed for the solution of schizophrenic statements [14,3,18]. In general, these solutions can be classified into those working at the source code level [18] and those working on the equation system [14,3].

3.1 Poigné and Holenderski's Solution

Poigné and Holenderski defined a translation of pure Esterel programs to Boolean equation systems [14]. Their translation also solved schizophrenia problems of local declarations without delayed actions. Given a statement S with locations (i.e., **pause** statements) $\ell_1, \ldots, \ell_l$, inputs $x_1,\ldots,x_m$, and outputs $y_1, \ldots, y_n$, they compute equations $\textbf{next}(\ell_i) = \varphi_i$ and $y_i = \psi_i$, where φ_i and ψ_i are propositional formulas in the variables x_i, y_i, and ℓ_i. For the remainder, the following definition is required, where we use again *st* as a special start location. Moreover, $[\tau]_x^e$ means replacement of all occurrences of x in τ by e:

Definition 3.1 *Given a term τ containing potential occurrences of the Boolean variables st, ℓ_1, ..., ℓ_n, we define the α-part $\alpha_L(\tau)$ as $\alpha_L(\tau) :\equiv [\tau]^{0\ldots 0}_{\ell_1\ldots\ell_n}$ with $L := \{\ell_1, \ldots, \ell_n\}$. Moreover, we define the η-part $\eta_{st}(\tau)$ of τ as $\eta_{st}(\tau) :\equiv [\tau]^{0}_{st}$.*

The intuition is thereby that $\alpha_L(\tau)$ equals to τ under the assumption that the control flow is currently not inside S. In particular, $\alpha_L(\tau)$ equals to τ when the control flow enters τ for the first time. Analogously, $\eta_{st}(\tau)$ equals to τ under the assumption that $st = 0$, i.e., when the statement is currently not started. In particular, $\eta_{st}(\tau)$ equals to τ when the control flow moves inside S, since we have the invariant that statements must not be (re)started if they are currently active and do not terminate. Note that the α- and η-parts of a term are not disjoint, which is the source of schizophrenia problems. For example, the α- and η-parts of the transition relation of the control flow $\mathcal{R}_{\mathsf{cf}}(st, S)$ are as follows:

- $\alpha_L(\mathcal{R}_{\mathsf{cf}}(st, S)) :\equiv \left(\begin{array}{l} st \wedge \mathsf{inst}\,(S) \wedge \neg\mathsf{next}(\mathsf{in}\,(S)) \vee \\ st \wedge \mathsf{enter}\,(S) \vee \\ \neg st \wedge \neg\mathsf{next}(\mathsf{in}\,(S)) \end{array}\right) \wedge \neg\mathsf{next}(st)$
- $\eta_{st}(\mathcal{R}_{\mathsf{cf}}(st, S)) :\equiv \left(\begin{array}{l} \neg\mathsf{in}\,(S) \wedge \neg\mathsf{next}(\mathsf{in}\,(S)) \vee \\ \mathsf{term}\,(S) \wedge \neg\mathsf{next}(\mathsf{in}\,(S)) \vee \\ \mathsf{move}\,(S) \end{array}\right) \wedge \neg\mathsf{next}(st)$

Although st is set to $\mathsf{0}$ in $\eta_{st}(\tau)$, we can not conclude from $\eta_{st}(\tau)$ that st is false, since st simply does not occur in $\eta_{st}(\tau)$. Moreover, it is easily seen that the case distinction made by $\alpha_L(\mathcal{R}_{\mathsf{cf}}(st, S))$ and $\eta_{st}(\mathcal{R}_{\mathsf{cf}}(st, S))$ is complete, i.e., that $\mathcal{R}_{\mathsf{cf}}(st, S) \Leftrightarrow \neg\mathsf{in}\,(S) \wedge \alpha_L(\mathcal{R}_{\mathsf{cf}}(st, S)) \vee \neg st \wedge \eta_{st}(\mathcal{R}_{\mathsf{cf}}(st, S))$ holds. Simply note that $st \wedge \neg\mathsf{in}\,(S)$ holds at starting time, and afterwards, we have $\neg st$.

The α- and η-parts of a term τ correspond to different views on τ that are made in the surface and the depth of a statement. Poigné and Holenderski's used this to rename locally declared variables in the surface part, i.e., in the α-parts of the right hand sides of their equation systems. Hence, they compute new equations $\mathbf{next}(\ell_i) = [\alpha_L(\varphi_i)]^{x'}_{x} \vee \eta_{st}(\varphi_i)$ and $y_i = [\alpha_L(\psi_i)]^{x'}_{x} \vee \eta_{st}(\psi_i)$, respectively, when the equation system of a local declaration has to be computed. As mentioned above, this renaming must be done for *all* local variables occurring in S, so that deeply nested local variables yield multiple copies.

The advantage of the approach is that it is remarkably simple and clear. However, the extensions to non-Boolean data types and delayed actions is unclear. Moreover, variables are even renamed if the local declaration is not nested in a loop, and therefore the procedure generates more copies than necessary.

3.2 Berry's Solution

Of course, the public domain Esterel compiler [4] and commercial tools like Esterel Studio [8] are also able to solve schizophrenia problems. Due to the different set of basic statements (traps instead of aborts), Berry also considers schizophrenic

parallel statements (which has also been done in [14]). The solution given in [3] defines for every statement its ‘incarnation level’, which is intuitively the number of necessary copies of its surface. This duplication of code segments is necessary to distinguish between different incarnations, and can not be circumvented. On the other hand, the procedure described in [3] is quite complicated, and therefore it is hard to extend it with optimizations, or to check its correctness, e.g., with a theorem prover. Moreover, similar to Poigné and Holenderski’s approach it is described only at the Boolean level and does not consider delayed actions.

3.3 Surface-Depth Splitting of Statements

In [18] a new approach to solve schizophrenia problems has been presented. Its main idea is to define for every statement S corresponding statements $\mathsf{surface}\,(S)$ and $\mathsf{depth}\,(S)$ such that $\mathsf{surface}\,(S)$ is that part of S that is executed when S is entered, and $\mathsf{depth}\,(S)$ is the remaining part of S. Both statements are defined in [18] by a simple primitive recursion over the statements, and can be computed in time $O(|S|^2)$. The reason for the quadratic blow-up is that sequences and loops generate copies of surface statements [18]. We do not consider the definitions of $\mathsf{surface}\,(S)$ and $\mathsf{depth}\,(S)$ here, but list the following result of [18]:

Theorem 3.2 (Surface and Depth) *For every statement S, we have:*

- $\mathsf{surface}\,(S)$ *is instantaneous for all inputs*
- S *and* $\mathsf{depth}\,(S)$ *have the same control flow*
- S *and* $\mathsf{surface}\,(S)\,;\mathsf{depth}\,(S)$ *have the same control flow*
- S *and* $\mathsf{surface}\,(S)\,;\mathsf{depth}\,(S)$ *have the same data flow*
- *no actions of* $\mathsf{depth}\,(S)$ *are executed when entering* $\mathsf{depth}\,(S)$

The idea proposed in [18] is then roughly as follows: we replace a local declaration **local** x **in** S **end** by the following statement, where $x^{(1)}$ is a copy of x with the same type:

$$\textbf{local}\ x, x^{(1)}\ \textbf{in}\ [\mathsf{surface}\,(S)]_x^{x^{(1)}}\,;\mathsf{depth}\,(S)\ \textbf{end}$$

The splitting of S into its surface and depth generates new occurrences of actions that definitely belong to either the surface or the depth, while in S, there may be actions that belong both to the surface and the depth. Hence, this splitting allows one to rename the local variable in the surface.

However, the above transformation is not sufficient. The splitting into surface and depth extracts all actions that are executed in the surface. However, if $\mathsf{depth}\,(S)$ contains a conditional statement whose condition is evaluated at starting time, then this evaluation should also refer to the surface values. However, simply renaming conditions of conditional statements is clearly wrong. Even more, there are statements with schizophrenic conditional statements, i.e., where one and the same condition is evaluated twice at one point of time, a first time with the surface values, and a second time with the depth values.

It has been proposed in [18] to replace such conditions φ by $\psi \wedge \varphi \vee \neg\psi \wedge [\varphi]_x^{x^{(1)}}$ using an expression ψ that holds exactly when S is entered. However, a condition may have several incarnations and hence, we must apply this transformation according to the number of possible incarnations.

In [18], it has moreover been erroneously stated that one copy of a local variable would be sufficient. In general, this is true, since only one surface proceeds to the depth and the other surface values are hidden, and can therefore be eliminated by the compiler. Nevertheless, the procedure listed in [18] is not correct, which has been pointed out by Edwards [7]. A correction of this procedure is possible, but due to different copies, the replacement of if-conditions as outlined above becomes quite complex.

As an alternative, one could define a new statement **goto** L, where L is a list of control flow locations. The semantics is that the control directly moves to the listed locations L to wait for the next macro step. Using such a statement, one could generate copies of the conditionals in the surface, so that their schizophrenia is also cured. A similar extension of Esterel with goto statements has been recently presented in [20]. Combining [18] and [20] gives another solution to schizophrenia at the statement level (see [20]).

4 The New Solution

Our new solution is based on various prerequisites that we have developed in previous work [15,16,18,17]. The main idea of the translation is to compute the control flow predicates inst (S), in (S), enter (S), move (S), term (S), and the guarded commands of a statement S, as defined in [16], in one recursion over S.

To this end, the translation has to forward starting conditions go^η and go^α during the recursive descent. The starting condition go^η enforces the control to enter the considered statement (if possible). The other condition go^α differs due to weak abortion: consider **weak abort** $S_1; S_2$ **when** kl, and assume the control is currently at a position in S_1 where S_1 terminates and assume that kl holds, so that the abortion takes place. As the abortion is weak, we still have to execute the actions of the surface of S_2, but the control must not enter S_2. Hence, the start condition go^α of S_2 holds, but its starting condition go^η is false. In general, we have $go^\eta \rightarrow go^\alpha$, but not vice versa. Moreover, the translation maintains conditions kl and sp for keeping track of surrounding abortion and suspension conditions.

The translation yields a tuple $(C, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$. C is thereby a set of new input variables that are used to mimic nondeterminism for choice and asynchronous concurrency (that can be controlled by a scheduler). L is the set of variables that are locally declared in S. We assume that all local variables have different names, hence, there is no shadowing. Moreover, names of local variables are different from names of input and output variables. I, A, and T are simply equivalent to inst (S), in (S), and term (S), respectively.

The main idea to solve schizophrenia problems is to split statements into their surface and depth. Hence, we have to split the control flow predicates and the guarded commands into surface and depth parts, respectively. However, it is easily seen that $\mathsf{inst}\,(S)$ and $\mathsf{enter}\,(S)$ refer to the surface and that $\mathsf{term}\,(S)$ and $\mathsf{move}\,(S)$ refer to the depth. Hence, there is no need to distinguish surface and depth parts of these components. In contrast, the guarded commands and the transition equations have to be split into surface parts G^α, R^α and depths parts G^η, R^η, respectively.

Therefore, the control flow is computed in form of two equation systems R^α and R^η that contain for every location variable ℓ_i a unique equation of the form $\mathsf{next}(\ell_i) = \varphi_i$. In principle, this is the same as hardware circuit generation, since hardware circuits are Boolean equation systems. Hence, we use similar templates as presented in [15] to this end (we use, however, some optimizations). As expected, R^α and R^η correspond to the surface and the depth of the control flow. Moreover, there is no (explicit) computation of $\mathsf{enter}\,(S)$ and $\mathsf{move}\,(S)$, since both are more or less the conjunction over R^α and R^η, respectively.

We made the experience that a lot of common subterms are generated by the translation. To obtain a translation that runs in quadratic time, we have to abbreviate common subterms. To this end, the translation maintains further equation systems E^α and E^η that contain the abbreviations made for the surface and the depth, respectively. Note that we have to distinguish between these equation systems due to possible renaming in the surface (see the discussion below). To summarize, our translation given in the appendix works as follows:

Theorem 4.1 (Correctness of Compile**)** *Given a* Quartz *statement S, a starting condition go^η, a precondition go^α, a suspension condition sp, and an abortion condition kl, the function call* $\mathsf{Compile}\,(go^\eta, go^\alpha, sp, kl, S)$ *of the function* Compile *implemented in Figures A.1-A.4 yields a tuple* $(C, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$ *with the following meaning:*

- *C is the set of added control variables* [2] *to eliminate nondeterministic statements (nondeterministic choice and asynchronous concurrency)*
- *L is the set of variables that are locally declared in S*
- $I = \mathsf{inst}\,(S)$
- $A = \mathsf{in}\,(S)$
- $T = \mathsf{term}\,(S)$
- *E^α are abbreviations to share common subterms of the surface*
- *E^η are abbreviations to share common subterms of the depth*
- $G^\alpha = \mathsf{guardcmd}\,(go^\alpha, \mathsf{surface}\,(S))$
- $G^\eta = \mathsf{guardcmd}\,(go^\alpha, \mathsf{depth}\,(S))$
- $\alpha_L(\mathcal{R}_{\mathsf{cf}}(go^\eta, S)) \Leftrightarrow \bigwedge_{\tau \in R^\alpha} \tau$
- $\eta_{go^\eta}(\mathcal{R}_{\mathsf{cf}}(go^\eta, S)) \Leftrightarrow \bigwedge_{\tau \in R^\eta} \tau$

[2] These are added as new inputs to mimic nondeterminism.

Initially, we use $go^\eta := go^\alpha := st$, $sp = kl = 0$ with the start location st whose transition equations $\textbf{init}(st) := 1$ and $\textbf{next}(st) := 0$ are furthermore added to the result. As Compile $(go^\eta, go^\alpha, sp, kl, S)$ computes equation systems R^α and R^η that implement the control flow of the surface and the depth, respectively, we have to combine them at the end:

Definition 4.2 *Given two transition systems R^α and R^η for the same state variables, we define the combined transition system $(R^\alpha \& R^\eta)$ as follows:*

$$(R^\alpha \& R^\eta) :\equiv \{[\mathsf{next}(\ell) = \tau_\alpha \vee \tau_\eta] \mid [\mathsf{next}(\ell) = \tau_\alpha] \in R^\alpha \vee [\mathsf{next}(\ell) = \tau_\eta] \in R^\eta\}$$

Simple set unions for the abbreviations E^α, E^η and the guarded commands G^α, G^η complete the translation: $E := E^\alpha \cup E^\eta$ and $G := G^\alpha \cup G^\eta$.

Finally, consider the *translation of local declarations: We want to rename local variables in those surfaces of their local declaration that can be executed at the same time with their depth.* Hence, we have to keep track of surfaces that contain surfaces of local declarations. To this end, the function Compile returns the set of locally declared variables. However, we still have to find the surfaces that must be renamed. Hence, consider how loops and sequences generate copies of surfaces [18] (the lines separates surface and depth of the entire statement):

- $S_1; S_2 :\equiv$
  ```
  (
    surface (S1) ;
    if inst (S1) then [surface (S2)] end;
    ----------------------------------------
    depth (S1) ;
    if in (S1) then surface (S2) end;
    depth (S2)
  )
  ```
- **do** S **while** $\sigma :\equiv$
  ```
  (
    [surface (S)]
    ----------------------------------------
    do
      depth (S) ;
      if σ then surface (S) end
    while σ
  )
  ```

Hence, a sequence $S_1; S_2$ generates two instances of surface (S_2), and a loop generates two instances of the surface of its body. Note that these instances can not be shared, since we may have to rename variables in one of them. In the algorithms of the appendix, we use a special function CompileSurface (Figure A.4) that works essentially like Compile, but only computes the surface parts (and therefore its runtime is linear). The reason for using this extra function is that the two instances of the surfaces have different start conditions that can be easily forwarded by a new function call.

Recall now that the reason for a schizophrenia problem is that the depth of a local declaration is executed at the same time with one of its surfaces. This may only happen when the local declaration is contained in a loop body. Hence, we only have to consider the two surfaces of a loop body above. Clearly, only the one in the if-statement can overlap with the depth (unless there is another surrounding

loop whose translation solves this additional overlapping problem). Hence, if a loop is compiled whose body contains local declarations of the variables L, then we rename these variables in the surface parts that belong to the surface of the if-statement above: Having compiled the body statement with a recursive call to Compile, the function Rename in Figure A.3 generates new variable names for the locally declared variables. The result is a substitution ϱ that maps old variables to new ones. Using this substitution [3], the local variables are renamed in the abbreviations E^α and the entering transitions R^α. Then, the guarded commands of the surface G^α are renamed by the following function RenameGuards:

- $\mathsf{RenameGuards}(\varrho, (\gamma, \textbf{now}\ \sigma)) := (\varrho(\gamma), \textbf{now}\ \varrho(\sigma))$
- $\mathsf{RenameGuards}(\varrho, (\gamma, \textbf{emit}\ x)) := (\varrho(\gamma), \textbf{emit}\ \varrho(x))$
- $\mathsf{RenameGuards}(\varrho, (\gamma, \textbf{emit next}(x)) := (\varrho(\gamma), \textbf{emit next}(x))$
- $\mathsf{RenameGuards}(\varrho, (\gamma, x := \tau)) := (\varrho(\gamma),\ \varrho(x) := \varrho(\tau))$
- $\mathsf{RenameGuards}(\varrho, (\gamma, \textbf{next}(x) := \tau)) := (\varrho(\gamma), \textbf{next}(x) := \varrho(\tau))$

Hence, only current values are substituted by this function, and delayed actions transfer the computed values to the depth. For this reason, the renaming is not sufficient: Additionally, we have to disable delayed actions on the local variables that would take place when the depth of the loop body is left. Otherwise, these action would transfer values to the next macro step of the new scope, which would contradict the semantics. This ‘disabling’ is done in the code for the loop with the function DisableDelayedLocals that is defined as $\mathsf{DisableDelayedLocals}(L, d, G) := \{\mathsf{DisableDL}(L, d, (\gamma, \alpha)) \mid (\gamma, \alpha) \in G\}$, where:

- $\mathsf{DisableDL}(L, d, (\gamma, \textbf{emit next}(x)) := \begin{cases} (\gamma, \textbf{emit next}(x)) & : x \notin L \\ (\gamma \wedge \neg d, \textbf{emit next}(x)) & : x \in L \end{cases}$
- $\mathsf{DisableDL}(L, d, (\gamma, \textbf{next}(x) := \tau)) := \begin{cases} (\gamma, \textbf{next}(x) := \tau) & : x \notin L \\ (\gamma \wedge \neg d, \textbf{next}(x) := \tau) & : x \in L \end{cases}$
- $\mathsf{DisableDL}(L, d, (\gamma, \mathcal{C}) := (\gamma, \mathcal{C})$ for all immediate actions $\mathcal{C}$

However, this is still not sufficient: Due to an abortion, it is possible that both instances of surface (S) are executed at the same time with the depth (S). Inspecting this situation shows that only one of these surfaces proceeds to the depth without being aborted, namely the boxed instance above. Hence, we additionally disable the delayed actions on local variables in the other instance if the boxed instance is also active. This is done by the first call to DisableDelayedLocals in the translation of loops.

This completes the translation. Note that we only rename when loops are encountered, in contrast to [14,18], where renaming is made in the translation of local declarations. Clearly, the algorithms in the appendix are not optimal, since they generate copies of *all contained* local variables when a loop is passed. A refined version should check if a reincarnation is possible by examining the satisfiability of the start and termination condition of the local declaration.

[3] We write $\varrho(\tau)$ for the expression that is obtained by replacing all occurrences of old variables by new ones according to ϱ.

References

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

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

[3] Berry, G., *The constructive semantics of pure Esterel*, http://www-sop.inria.fr/esterel.org (1999).

[4] Berry, G., *The Esterel v5_91 language primer* (2000).

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

[6] Brzozowski, J. and C.-J. Seger, "Asynchronous Circuits," Springer, 1995.

[7] Edwards, S., *Personal communications* (2002).

[8] Esterel Technology, *Website*, http://www.esterel-technologies.com.

[9] Girault, A. and G. Berry, *Circuit generation and verification of Esterel programs*, in: *Symposium on Signals, Circuits, and Systems (SCS)*, Iasi, Romania, 1999, pp. 85–89.

[10] Gordon, M. and T. Melham, "Introduction to HOL: A Theorem Proving Environment for Higher Order Logic," Cambridge University Press, 1993.

[11] Halbwachs, N. and F. Maraninchi, *On the symbolic analysis of combinational loops in circuits and synchronous programs*, in: *Euromicro Conference*, Como, Italy, 1995.

[12] Lavagno, L. and E. Sentovich, *ECL: A specification environment for system-level design*, in: *Design Automation Conference (DAC)* (1999).

[13] Malik, S., *Analysis of cycle combinational circuits*, IEEE Transactions on Computer Aided Design **13** (1994), pp. 950–956.

[14] Poigné, A. and L. Holenderski, *Boolean automata for implementing pure Esterel*, Arbeitspapiere 964, GMD, Sankt Augustin (1995).

[15] Schneider, K., *A verified hardware synthesis for Esterel*, in: F. Rammig, editor, *Workshop on Distributed and Parallel Embedded Systems (DIPES)* (2000), pp. 205–214.

[16] Schneider, K., *Embedding imperative synchronous languages in interactive theorem provers*, in: *Conference on Application of Concurrency to System Design (ICACSD)* (2001), pp. 143–156.

[17] Schneider, K., *Proving the equivalence of microstep and macrostep semantics*, in: *Conference on Theorem Proving in Higher Order Logic*, LNCS **2410** (2002), pp. 314–331.

[18] Schneider, K. and M. Wenz, *A new method for compiling schizophrenic synchronous programs*, in: *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (2001), pp. 49–58.

[19] Shiple, T., G. Berry and H. Touati, *Constructive analysis of cyclic circuits*, in: *European Design and Test Conference (EDTC)* (1996).

[20] Tardieu, O., *Goto and concurrency: Introducing safe jumps in Esterel*, in: *Synchronous Languages, Applications, and Programming (SLAP)*, ENTCS (2004).

A Implementation

function Compile $(go^\eta, go^\alpha, sp, kl, P)$
case P **of**
nothing :
return $(\{\}, \{\}, 1, 0, 0, \{\}, \{\}, \{\}, \{\}\{\}, \{\})$;
emit x, **emit next**(x), **now** σ, $y := \tau$, **next**$(y) := \tau$:
return $(\{\}, \{\}, 1, 0, 0, \{\}, \{\}, \{(go^\alpha, P)\}, \{\}, \{\}, \{\})$;
ℓ : **pause** :
return $(\{\}, \{\}, 0, \ell, \ell, \{\}, \{\}, \{\}, \{\}, \{\mathsf{next}(\ell) = go^\eta\}, \{\mathsf{next}(\ell) = sp \wedge \ell\})$;
if σ **then** S_1 **else** S_2 **end** :
return cond_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S_1, S_2)$;
choose S_1 [] S_2 **end** :
$c :=$ **newvar**();
$(C, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$
$:=$ cond_Compile $(go^\eta, go^\alpha, sp, kl, c, S_1, S_2)$;
return $(C \cup \{c\}, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$;
$S_1; S_2$: **return** seq_Compile $(go^\eta, go^\alpha, sp, kl, S_1, S_2)$;
$S_1 \parallel S_2$: **return** par_Compile $(go^\eta, go^\alpha, sp, kl, S_1, S_2)$;
$S_1 ||| S_2$:
$c_1 :=$ **newvar**(); $P_1 :=$ **mk_suspend**$(0, S_1, \neg c_1)$;
$c_2 :=$ **newvar**(); $P_2 :=$ **mk_suspend**$(0, S_2, \neg c_2)$;
$\sigma := [\mathsf{in}\,(S_1) \wedge c_1] \vee [\mathsf{in}\,(S_2) \wedge c_2]$;
$P :=$ **mk_during**(**mk_par**$(P_1, P_2), \sigma)$;
$(C, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta) :=$ Compile $(go^\eta, go^\alpha, sp, kl, P)$;
return $(C \cup \{c_1, c_2\}, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$;
do S **while** σ : **return** dowhile_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S)$;
suspend S **when** σ : **return** susp_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S, 0)$;
weak suspend S **when** σ : **return** susp_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S, 1)$;
abort S **when** σ : **return** abort_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S, 0)$;
weak abort S **when** σ : **return** abort_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S, 1)$;
local x **in** S **end**, **local** $x : \alpha$ **in** S **end** :
$(C, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta) :=$ Compile $(go^\eta, go^\alpha, sp, kl, S)$;
return $(C, L \cup \{x\}, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$;
during S **holds** σ :
$(C, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta) :=$ Compile $(go^\eta, go^\alpha, sp, kl, S)$;
return $(C, L, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta \cup \{(A, \textbf{now}\ \sigma)\}, R^\alpha, R^\eta)$;
end case ;
end function

Figure A.1. Translation of Quartz Statements (part I)

function seq_Compile $(go^\eta, go^\alpha, sp, kl, S_1, S_2)$
$(C_1, L_1, I_1, A_1, T_1, E_1^\alpha, E_1^\eta, G_1^\alpha, G_1^\eta, R_1^\alpha, R_1^\eta) :=$ Compile $(go^\eta, go^\alpha, sp, kl, S_1)$;
$go_2^\eta :=$ **newvar**(); $go_2^\alpha :=$ **newvar**();
$E^\alpha := \{(go_2^\eta := go^\eta \wedge I_1), (go_2^\alpha := go^\alpha \wedge I_1)\}$;
$(I_2, E_2^\alpha, G_2^\alpha, R_2^\alpha) :=$ CompileSurface $(go_2^\eta, go_2^\alpha, S_2)$;
$go_3^\eta :=$ **newvar**();
$E^\eta := \{go_3^\eta := T_1 \wedge \neg(sp \vee kl)\}$;
$(C_3, L_3, I_3, A_3, T_3, E_3^\alpha, E_3^\eta, G_3^\alpha, G_3^\eta, R_3^\alpha, R_3^\eta) :=$ Compile $(go_3^\eta, T_1, sp, kl, S_2)$;
$E^\alpha := E_2^\alpha \cup E_1^\alpha \cup E^\alpha$; $E^\eta := E_3^\alpha \cup E_3^\eta \cup E_1^\eta \cup E^\eta$;
$G^\alpha := G_2^\alpha \cup G_1^\alpha$; $G^\eta := G_1^\eta \cup G_3^\alpha \cup G_3^\eta$;
$R^\alpha := R_2^\alpha \cup R_1^\alpha$; $R^\eta := R_1^\eta \cup (R_3^\alpha \& R_3^\eta)$;
$I :=$ **newvar**(); $A :=$ **newvar**(); $T :=$ **newvar**();
$E^\alpha := \{I := I_1 \wedge I_2\} \cup E^\alpha$;
$E^\eta := \{(A := A_1 \vee A_2), (T := T_1 \wedge I_2 \vee T_2)\} \cup E^\eta$;
return $(C_1 \cup C_2, L_1 \cup L_2, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$;
end function

function par_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S_1, S_2)$
$(C_1, L_1, I_1, A_1, T_1, E_1^\alpha, E_1^\eta, G_1^\alpha, G_1^\eta, R_1^\alpha, R_1^\eta) :=$ Compile $(go^\eta, go^\alpha, sp, kl, S_1)$;
$(C_2, L_2, I_2, A_2, T_2, E_2^\alpha, E_2^\eta, G_2^\alpha, G_2^\eta, R_2^\alpha, R_2^\eta) :=$ Compile $(go^\eta, go^\alpha, sp, kl, S_2)$;
$I :=$ **newvar**(); $A :=$ **newvar**(); $T :=$ **newvar**();
$E^\alpha := \{I := I_1 \wedge I_2\} \cup E_1^\alpha \cup E_2^\alpha$;
$E^\eta := \{A := A_1 \vee A_2\} \cup E_1^\eta \cup E_2^\eta$;
$E^\eta := \{T := T_1 \wedge \neg A_2 \vee T_2 \wedge \neg A_1 \vee T_1 \wedge T_2\} \cup E^\eta$;
$G^\alpha := G_1^\alpha \cup G_2^\alpha$; $G^\eta := G_1^\eta \cup G_2^\eta$;
$R^\alpha := R_1^\alpha \cup R_2^\alpha$; $R^\eta := R_1^\eta \cup R_2^\eta$;
return $(C_1 \cup C_2, L_1 \cup L_2, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$
end function

function dowhile_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S)$
$(I_1, E_1^\alpha, G_1^\alpha, R_1^\alpha) :=$ CompileSurface (go^η, go^α, S);
$go_2^\eta :=$ **newvar**(); $go_2^\alpha :=$ **newvar**(); $T :=$ **newvar**();
$(C, L_2, I_2, A_2, T_2, E_2^\beta, E_2^\eta, G_2^\beta, G_2^\eta, R_2^\beta, R_2^\eta) :=$ Compile $(go_2^\eta, go_2^\alpha, sp, kl, S)$;
$G_2^\beta :=$ DisableDelayedLocals$(L_2, go^\alpha, G_2^\beta)$;
$G_2^\eta :=$ DisableDelayedLocals$(L_2, go^\alpha, G_2^\eta)$;
$(\varrho, E_2^\beta, G_2^\beta, R_2^\beta) :=$ Rename$(L_2, E_2^\beta, G_2^\beta, R_2^\beta)$;
$E^\eta := \{go_2^\eta := go_2^\alpha \wedge \neg(sp \vee kl), go_2^\alpha := T_2 \wedge \sigma\} \cup E_2^\beta \cup E_2^\eta$;
$G^\eta := G_2^\beta \cup G_2^\eta$;
$R^\eta := \left(R_2^\beta \& R_2^\eta\right)$;
$T :=$ **newvar**(); $E^\eta := \{T := T_1 \wedge \neg\sigma\} \cup E^\eta$;
return $(C, L_2, 0, A_2, T, E_1^\alpha, E^\eta, G_1^\alpha, G^\eta, R_1^\alpha, R^\eta)$;
end function

Figure A.2. Translation of Quartz Statements (part II)

function Rename$(L, E^\alpha, G^\alpha, R^\alpha)$
$\varrho := \{(x, \mathbf{newvar}()) \mid x \in L\}$;
$E^\alpha := \{\varrho(\tau) \mid \tau \in E^\alpha\}$;
$R^\alpha := \{\varrho(\tau) \mid \tau \in R^\alpha\}$;
$G^\alpha := \{\mathsf{RenameGuards}(\varrho, (\gamma, \alpha)) \mid (\gamma, \alpha) \in G^\alpha\}$;
return $(\varrho, E^\alpha, G^\alpha, R^\alpha)$;
end function

function susp_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S, wk)$
$sp_1 := \mathbf{newvar}()$; $E^\beta := \{sp_1 := sp \vee \sigma \wedge \neg kl\}$;
$(C, L, I, A, T_1, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta) :=$ Compile $(go^\eta, go^\alpha, sp_1, kl, S)$;
$T := \mathbf{newvar}()$; $E^\beta := \{T := T_1 \wedge \neg sp_1\} \cup E^\beta$;
if $\neg wk$ **then** $G^\eta := \{(\gamma \wedge \neg\sigma, \alpha) \mid (\gamma, \alpha) \in G^\eta\}$ **end**;
return $(C, L, I, A, T, E^\alpha, E^\eta \cup E^\beta, G^\alpha, G^\eta, R^\alpha, R^\eta)$;
end function

function abort_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S, wk)$
$kl_1 := \mathbf{newvar}()$; $E^\beta := \{kl_1 := kl \vee \sigma\}$;
$(C, L, I, A, T_1, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta) :=$ Compile $(go^\eta, go^\alpha, sp, kl_1, S)$;
$T := \mathbf{newvar}()$; $E^\beta := \{T := T_1 \vee A \wedge kl_1\} \cup E^\beta$;
if $\neg wk$ **then** $G^\eta := \{(\gamma \wedge \neg\sigma, \alpha) \mid (\gamma, \alpha) \in G^\eta\}$ **end**;
return $(C, L, I, A, T, E^\alpha, E^\eta \cup E^\beta, G^\alpha, G^\eta, R^\alpha, R^\eta)$;
end function

function cond_Compile $(go^\eta, go^\alpha, sp, kl, \sigma, S_1, S_2)$
$go_1^\eta := \mathbf{newvar}()$; $go_1^\alpha := \mathbf{newvar}()$; $go_2^\eta := \mathbf{newvar}()$; $go_2^\alpha := \mathbf{newvar}()$;
$E^\alpha := \{(go_1^\eta := go^\eta \wedge \sigma), (go_1^\alpha := go^\alpha \wedge \sigma)\}$;
$E^\alpha := \{(go_2^\eta := go^\eta \wedge \neg\sigma), (go_2^\alpha := go^\alpha \wedge \neg\sigma)\} \cup E^\alpha$;
$(C_1, L_1, I_1, A_1, T_1, E_1^\alpha, E_1^\eta, G_1^\alpha, G_1^\eta, R_1^\alpha, R_1^\eta) :=$ Compile $(go_1^\eta, go_1^\alpha, sp, kl, S_1)$;
$(C_2, L_2, I_2, A_2, T_2, E_2^\alpha, E_2^\eta, G_2^\alpha, G_2^\eta, R_2^\alpha, R_2^\eta) :=$ Compile $(go_2^\eta, go_2^\alpha, sp, kl, S_2)$;
$A := \mathbf{newvar}()$; $T := \mathbf{newvar}()$; $I := \mathbf{newvar}()$;
$E^\alpha := \{I := \sigma \wedge I_1 \vee \neg\sigma \wedge I_2\} \cup E^\alpha \cup E_1^\alpha \cup E_2^\alpha$;
$E^\eta := \{(A := A_1 \vee A_2), (T := T_1 \vee T_2)\} \cup E_1^\eta \cup E_2^\eta$;
$G^\alpha := G_1^\alpha \cup G_2^\alpha$; $G^\eta := G_1^\eta \cup G_2^\eta$;
$R^\alpha := R_1^\alpha \cup R_2^\alpha$; $R^\eta := R_1^\eta \cup R_2^\eta$;
return $(C_1 \cup C_2, L_1 \cup L_2, go^\eta, go^\alpha, I, A, T, E^\alpha, E^\eta, G^\alpha, G^\eta, R^\alpha, R^\eta)$;
end function

Figure A.3. Translation of Quartz Statements (part III)

function CompileSurface (go^η, go^α, P)
  case P **of**
    nothing :
      return $(1, \{\}, \{\}, \{\})$;
    emit x, **emit next**(x):
      return $(1, \{\}, \{(go^\alpha, P)\}, \{\})$;
    $y := \tau$, **next**$(y) := \tau$:
      return $(1, \{\}, \{(go^\alpha, P)\}, \{\})$;
    now σ :
      return $(1, \{\}, \{(go^\alpha, P)\}, \{\})$;
    ℓ : **pause** :
      return $(0, \{\}, \{\}, \{\mathsf{next}(\ell) = go^\eta\})$;
    if σ **then** S_1 **else** S_2 **end** :
      $go_1^\eta :=$ **newvar**(); $go_1^\alpha :=$ **newvar**(); $go_2^\eta :=$ **newvar**(); $go_2^\alpha :=$ **newvar**();
      $E^\alpha := \{(go_1^\eta := go^\eta \wedge \sigma), (go_1^\alpha := go^\alpha \wedge \sigma)\}$;
      $E^\alpha := \{(go_2^\eta := go^\eta \wedge \neg\sigma), (go_2^\alpha := go^\alpha \wedge \neg\sigma)\} \cup E^\alpha$;
      $(I_1, E_1^\alpha, G_1^\alpha, R_1^\alpha) :=$ CompileSurface $(go_1^\eta, go_1^\alpha, S_1)$;
      $(I_2, E_2^\alpha, G_2^\alpha, R_2^\alpha) :=$ CompileSurface $(go_2^\eta, go_2^\alpha, S_2)$;
      $I :=$ **newvar**(); $E^\alpha := \{I := I_1 \wedge \sigma \vee I_2 \wedge \neg\sigma\} \cup E^\alpha$;
      return $(I, E_1^\alpha \cup E_2^\alpha \cup E^\alpha, G_1^\alpha \cup G_2^\alpha, R_1^\alpha \cup R_2^\alpha)$;
    $S_1; S_2$:
      $(I_1, E_1^\alpha, G_1^\alpha, R_1^\alpha) :=$ CompileSurface $(go^\eta, go^\alpha, S_1)$;
      $go_2^\eta :=$ **newvar**(); $go_2^\alpha :=$ **newvar**();
      $E^\alpha := \{(go_2^\eta := go^\eta \wedge I_1), (go_2^\alpha := go^\alpha \wedge I_1)\}$;
      $(I_2, E_2^\alpha, G_2^\alpha, R_2^\alpha) :=$ CompileSurface $(go_2^\eta, go_2^\alpha, S_2)$;
      $I :=$ **newvar**(); $E^\alpha := \{I := I_1 \wedge I_2\} \cup E^\alpha$;
      return $(I, E_1^\alpha \cup E_2^\alpha \cup E^\alpha, G_1^\alpha \cup G_2^\alpha, R_1^\alpha \cup R_2^\alpha)$;
    $S_1 \parallel S_2$:
      $(I_1, E_1^\alpha, G_1^\alpha, R_1^\alpha) :=$ CompileSurface $(go^\eta, go^\alpha, S_1)$;
      $(I_2, E_2^\alpha, G_2^\alpha, R_2^\alpha) :=$ CompileSurface $(go^\eta, go^\alpha, S_2)$;
      $I :=$ **newvar**(); $E^\alpha := \{I := I_1 \wedge I_2\}$;
      return $(I, E_1^\alpha \cup E_2^\alpha \cup E^\alpha, G_1^\alpha \cup G_2^\alpha, R_1^\alpha \cup R_2^\alpha)$;
    do S **while** σ : **return** CompileSurface (go^η, go^α, S);
    suspend S **when** σ : **return** CompileSurface (go^η, go^α, S);
    weak suspend S **when** σ : **return** CompileSurface (go^η, go^α, S);
    abort S **when** σ : **return** CompileSurface (go^η, go^α, S);
    weak abort S **when** σ : **return** CompileSurface (go^η, go^α, S);
    local x **in** S **end**, **local** $x : \alpha$ **in** S **end** : **return** CompileSurface (go^η, go^α, S);
    during S **holds** σ : **return** CompileSurface (go^η, go^α, S);
  end case ;
end function

Figure A.4. Computing only the Surface Parts