Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 153 (2006) 71–97
www.elsevier.com/locate/entcs

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

Abstract

We describe the translation of Esterel-like programs with delayed actions to equivalent transition relations and 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 these copies to allow them to have different values at the same point of time. The translation runs in polynomial time and has been formally verified with the HOL theorem prover.

Keywords: synchronous languages, verified compilers

1 Introduction

Synchronous languages [1] like Esterel [2,4] and its variants [13,17] offer an adequate programming paradigm for the development of reactive real-time systems. Several success stories have been reported [9] from safety-critical applications like avionics, automotive industries, transportation, and many others. The formal semantics of these languages allows us to apply formal methods not only to verify particular programs, but also to reason about the languages' semantics at a meta-level. In particular, this allows us to verify program transformations and the entire compilation [22,18] so that formally verified code generators are obtained.

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

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2006.02.028

time. Consumption of time is explicitly programmed by grouping finitely many micro steps to macro steps. As a consequence, all threads of the program run in lockstep: they execute the micro steps of the current macro step in zero time, and automatically synchronize at the end of the macro step. As all micro steps of a macro step are executed at the same point of time, their ordering within the macro step is irrelevant. Therefore, values of variables are determined with respect to macro steps instead of micro steps.

The abstraction to macro steps is the basic paradigm of the semantics of synchronous languages. It yields a clear programming model for multi-threaded programs and hardware circuits. However, this abstraction is not for free: Cyclic dependencies (causality cycles) and schizophrenia problems are the two major problems that must be solved by the compilers. *Causality cycles* arise when the condition for executing an action is immediately influenced by the result of 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 [14,12,6,25,5,3,19,21]. Usually, causality analysis is performed as a second step after the compilation to intermediate data structures.

Schizophrenia problems [15,3,22] have to be 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, but in different scopes. To avoid confusion, the compiler has to generate copies (incarnations) of locally declared variables. In general, several copies may be necessary since deeply nested abort and loop statements can enforce several executions of a local declaration in a single macro step. Referring to the right incarnations of local variables poses a difficult problem for the compilers, since all variables must have a uniquely determined value at each point of time (at least for hardware synthesis).

In this paper, we present a new algorithm for the translation of synchronous programs to a simple intermediate format: The control flow is given as an equation system [10,3,16] and the data flow is given as a set of guarded commands [7,17]. To solve schizophrenia problems, we have to compute control and data flow representations *separately for the surface and depth of a program* [15,3,17,22,27]. 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 and proceeds with its execution. Overlapped execution of surface and depth parts of local declarations can lead to schizophrenia problems since these micro steps belong to different scopes of the local variable. For this reason, we have to compute the control and data flow separately for the surface and depth of a statement, so that we are able to rename local variables in the surface parts. Furthermore, it is worth mentioning that it is sufficient to handle this issue during the compilation of loops (and not of local declarations), since loops are responsible for the schizophrenia problem.

We have embedded [17] our Esterel variant Quartz in the HOL theorem prover [11]. 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 [16] and the equivalence to SOS rules [18]. 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 the differences between Esterel and Quartz [16,17,18,22]. 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 [16,17,18] is a variant of Esterel [2,4,9] that extends classic Esterel by delayed assignments, delayed emissions, asynchronous concurrency, nondeterministic choice, and inline assertions (in the meantime, delayed actions are also available in Esterel v7, and oracle inputs allow one to implement the other statements, too). 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)*

There are two kinds of variables in Quartz, namely *event variables* and *state variables*, which are manipulated by emit and assignment statements, respectively [1]. State variables 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 a variable in the current macro step and are therefore rather equations than assignments.

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

In the following, assignments and emissions are called *actions* which can be either delayed or immediate. Delayed actions are particularly useful to describe hardware circuits. The additional *pre* operator of Esterel for accessing previous values is not used in Quartz (and neither for registered variables in Esterel v7).

Nondeterministic choice and asynchronous concurrency both introduce nondeterminism, which is usually not wanted in the development of react-

[1] Event variables of Quartz are called pure signals in Esterel, and state variables of Quartz resemble valued Esterel signals without a status. In Esterel v7 [9], state variables appear as value-only signals, that have to be declared with the `reg` keyword (as so-called registered variables) to allow the use of delayed actions. In Quartz, there are no variables in the sense of Esterel's local variables that can be manipulated by a single thread during micro steps (these variables can be easily eliminated with local (Quartz) variables with delayed assignments).

ive systems. Nondeterministic statements are implemented by unobservable (oracle) inputs [17] that may be controlled deterministically by an explicit scheduler. Hence, they may be of interest to offer the compiler some freedom for adding a suitable scheduler (e.g. to finally replace ||| with ||).

Immediate assertions of the form *now* σ require that σ currently holds, and *during* S *holds* σ requires that σ holds whenever the control is currently inside S [17]. 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 [17,16,18] and, in particular, 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 may depend on the input variables.

There is only one basic statement where the control can rest for the next macro step, namely the *pause* statement [2]. 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 a statement S is defined by the control flow conditions $\mathsf{in}(S)$, $\mathsf{inst}(S)$, $\mathsf{enter}(S)$, $\mathsf{term}(S)$, and $\mathsf{move}(S)$, and the data flow of S is defined by the set of guarded commands $\mathsf{guardcmd}(\varphi, S)$ [17]:

$\mathsf{in}(S)$ 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.

[2] To be precise, immediate forms of suspend also have this ability. Hence, they can replace *pause* as follows:

$$pause :\equiv \left[\begin{array}{l} abort \\ \quad suspend \\ \quad\quad nothing \\ \quad when\ immediate\ \mathsf{true} \\ when\ \mathsf{true} \end{array}\right]$$

$\mathsf{term}\,(S)$ 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 of the form *now* σ. The meaning of $(\gamma, \mathcal{C})$ is that $\mathcal{C}$ is immediately executed whenever its guard γ holds.

The above control flow conditions as well as the guarded commands can be defined by primitive recursion [17,18] over the statement. The definition of these conditions 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 then defined as follows [3]:

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

The four disjuncts of the transition relation describe 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 [17,22] and, in particular, in the appendix of this article.

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 thereby that a local declaration can be left and (re)entered within the same macro step. The micro steps of such a macro step must then refer to the right incarnation of the local variable, depending on whether they belong to the old or the new scope of the local declaration.

[3] It has been proved in [16,17] that $\mathcal{R}_{\mathsf{cf}}(st, S)$ is equivalent to an equation system that consists of one equation for each program location. In the appendix, we show how this equation system is computed, which is the basis for a translation to hardware circuits.

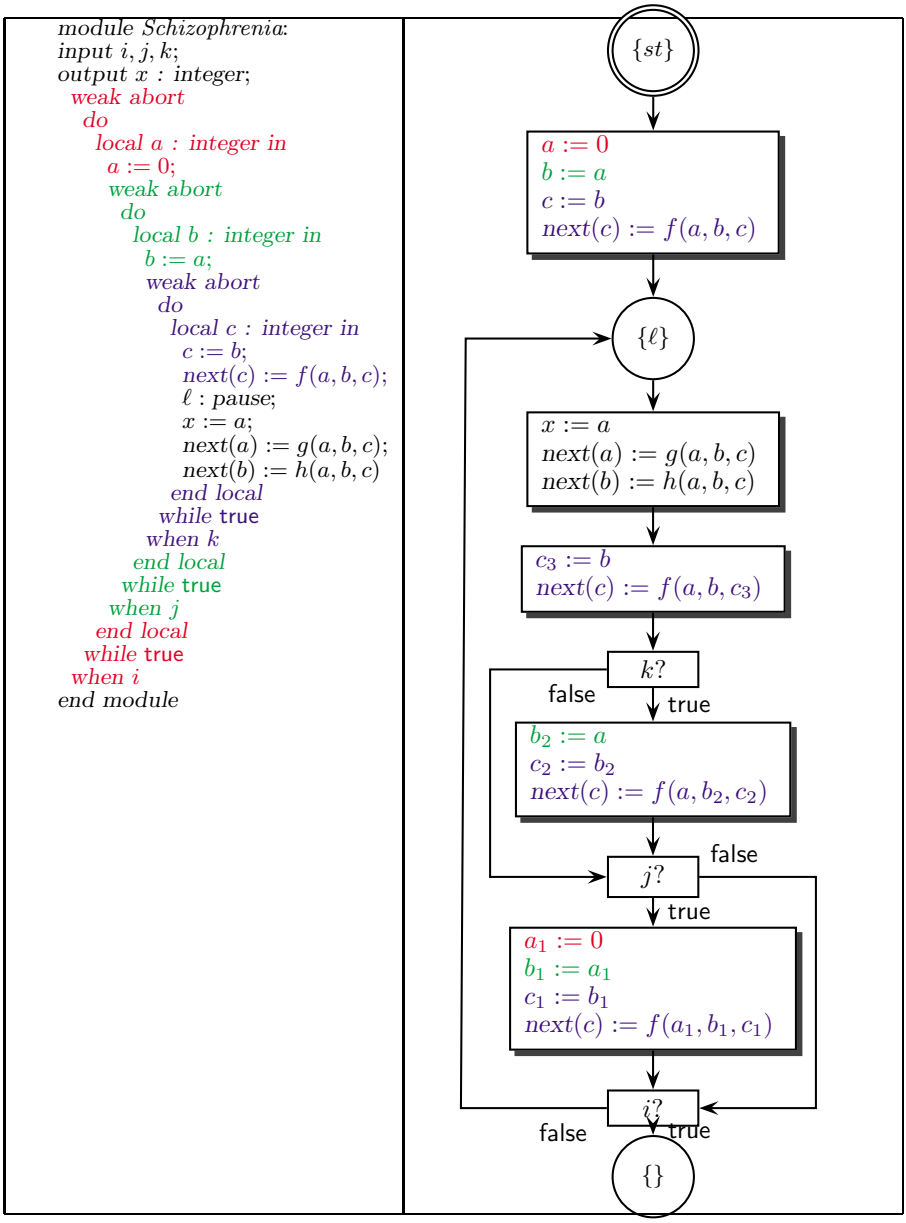

Figure 1. Local Declarations with Delayed Actions.

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 the corresponding control-data-flow graph. The circle nodes of this graph are control flow states that are labelled with those location variables that are currently active (including the start

location 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 branches that influence the following computations. The outgoing arcs of such a 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 action node belongs to the depth of all local declarations, the second one (re)enters the local declaration of c, but remains inside the local declarations of b and a. A new incarnation c_3 is thereby created. The node below condition node k? (re)enters the local declarations of b and c, but remains in the one of a. Hence, it creates new incarnations b_2 and c_2 of b and c, respectively. Finally, the remaining node, (re)enters all local declarations, and therefore generates three incarnations a_1, b_1, and c_1. Note that these four action boxes can be executed at the same point of time, and therefore, the reincarnations a_1, b_1, c_1, b_2, c_2, and c_3 may all exist in one macro step.

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 locally declared variables according to the number of the possible '(re)enterings' (which are also called surfaces).

Delayed actions add further difficulties to the reincarnation of locally declared variables: If a delayed action that changes the value of a 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 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 $next(c) := f(a, b, c)$, $next(c) := f(a, b, c_3)$, $next(c) := f(a, b_2, c_2)$, and

$next(c) := f(a_1, b_1, c_1)$ must be executed. In the code given in the appendix, we apply the disabling function only to the actions of the depth. However, note that the surface of a substatement may belong to the depth of a surrounding statement, and therefore, the disabling will also be handled in surfaces when required.

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 [15,3,22,27]. In general, these solutions can be classified into those working at the source code level [22,27] and those working on the equation system [15,3].

3.1 Poigné and Holenderski's Solution

Poigné and Holenderski defined a translation of pure Esterel programs (programs where all actions are immediate emissions) to Boolean equation systems [15]. Their translation also solved schizophrenia problems of local declarations without delayed actions. Given a statement S with locations $\ell_1, \ldots, \ell_l$, inputs $x_1,\ldots,x_m$, and outputs $y_1, \ldots, y_n$, they compute equations $next(\ell_i) = \varphi_i$ 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 substitution 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 $\alpha_L(\tau) :\equiv [\tau]_{\ell_1\ldots\ell_n}^{\mathsf{false}\ldots\mathsf{false}}$ with $L := \{\ell_1, \ldots, \ell_n\}$. Moreover, we define the η-part $\eta_{st}(\tau)$ of τ as $\eta_{st}(\tau) :\equiv [\tau]_{st}^{\mathsf{false}}$.*

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 $st = \mathsf{false}$, 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. In general, the α- and η-parts of the transition relation of the control flow $\mathcal{R}_{\mathsf{cf}}(st, S)$ are as follows:

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

Although st is set to false in $\eta_{st}(\tau)$, we can not conclude from $\eta_{st}(\tau)$ that st is false, since st simply does no longer 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 $next(\ell_i) = [\alpha_L(\varphi_i)]_x^{x'} \vee \eta_{st}(\varphi_i)$ 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 generalization 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 [9] are 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 [15]). The solution given in [3] (Chapters 12 and 13) 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 given at the circuit gate level, and therefore, it is quite complicated. Moreover, it is difficult to extend it with optimizations, and to check its correctness, e.g., with a theorem prover. Finally, 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 [22], 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 [22] 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 [22]. We do not consider the definitions of $\mathsf{surface}\,(S)$ and $\mathsf{depth}\,(S)$ here, but list the following result of [22]:

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 [22] 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:

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

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 (depending on the current values of the variables). 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.

For this reason, it has been proposed in [22] to replace such conditions φ by expressions like $\psi \wedge \varphi \vee \neg\psi \wedge [\varphi]_x^{x^{(1)}}$ 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 [22], 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 while the other surface values are hidden and can therefore be eliminated by the compiler. Nevertheless, the procedure listed in [22] is not correct, which has been pointed out by Edwards [8]. A correction of this procedure is possible, but due to different copies, the replacement of if-conditions as outlined above becomes quite difficult.

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. Such an extension of Esterel with goto statements has been recently presented in [26]. Combining [22] and [26] gives another solution to schizophrenia at the source code level (see [27]).

4 Our Verified Compilation Scheme

Our new solution is based on various prerequisites that we have developed in previous work [16,17,22,18]. Similar to [22], the main idea is to compute in one pass over a given statement S the required information like the control flow conditions $\mathsf{inst}(S)$, $\mathsf{in}(S)$, $\mathsf{enter}(S)$, $\mathsf{move}(S)$, $\mathsf{term}(S)$, and the guarded commands [17]. To this end, the translation has to forward start conditions go^α and go^η during the recursive descent: go^α enables the actions of the surface, and go^η additionally enforces the control to enter the considered statement [4]. In addition to go^α and go^η, the translation maintains the conditions kl and sp for keeping track of surrounding abortion and suspension conditions [5]. Moreover, a list of renamings Υ of the local variables is maintained

[4] In general, go^η implies go^α, but not vice versa: 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 further that kl holds, so that the abortion takes place. As the abortion is weak, we have to execute the actions of the surface of S_2, but the control must not enter S_2. Hence, the start condition go^α of S_2 holds, but go^η is false.

[5] kl has thereby higher priority, so we actually translate *abort suspend* S *when* sp *when* kl.

whose length depends on the maximal number of nested loops. We explain the latter below in more detail.

The essential key to solve schizophrenia problems is to consider surface and depth parts of statements and of related conditions like the control flow conditions and the guarded commands. It is easily seen that $\mathsf{inst}(S)$ and $\mathsf{enter}(S)$ completely refer to the surface, whereas $\mathsf{term}(S)$ and $\mathsf{move}(S)$ completely refer to the depth. Hence, there is no need to distinguish between surface and depth parts of these conditions. 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.

The surface and depth parts are computed by the functions CompileSurface and CompileDepth as shown in detail in the appendix. These functions are initially called by function StartCompile which also precomputes the required renamings of local variables Υ. The results of the function calls $\mathsf{CompileSurface}(\rho, go^\alpha, go^\eta, S)$ and $\mathsf{CompileDepth}(\Upsilon, sp, kl, S)$ are tuples (I, G^α, R^α) and $(C, L, \Xi, A, T, G^\eta, R^\eta)$ with the following meaning (see Theorem 4.2): C is a set of new (oracle) input variables that are used to mimic nondeterminism for choice and asynchronous concurrency (these variables can be controlled by an additional scheduler). The conditions I, A, and T are simply the control flow predicates $\mathsf{inst}(S)$, $\mathsf{in}(S)$, and $\mathsf{term}(S)$, respectively. G^α and G^η are the sets of guarded commands of the surface and depth of S, respectively. Moreover, R^α and R^η are equation systems [6] that contain for every location variable ℓ_i a unique equation of the form $\mathsf{next}(\ell_i) = \varphi_i$. R^α and R^η are the α- and η-parts of the transition equations. It is interesting to note that the conjunction of R^α is equivalent to $\mathsf{enter}(S)$.

Finally, L is the set of variables that are locally declared in S (we assume that there are no shadowing problems), and Ξ is a list that contains for each reincarnated surface of loops in S a set of tuples (go^η, x, x') such that x' is the renaming of a local variable $x \in L$ that is used in the surface with start condition go^η. In Appendix B, it is explained why this information is required for finally generating a transition relation or executable code from the intermediary results.

The final result, as determined by the function StartCompile, simply computes $G^\alpha \cup G^\eta$ and $(R^\alpha \& R^\eta)$ as representations of the data and control flow, where $(R^\alpha \& R^\eta)$ is defined as follows:

[6] In principle, such equation systems are nothing else but hardware circuits. Hence, we use similar templates as presented in [16] for their computation.

Definition 4.1 *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:*

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

To obtain a translation that runs in quadratic time, subterms that are generated during the translation have to be abbreviated by new variables. In the proceedings version of this paper, we used two equation systems E^α and E^η that contained the abbreviations made for the surface and the depth, respectively. This was necessary since the solution of schizophrenia problems requires to rename the locally declared variables in the surfaces of loops (see below). Hence, we had to split the equation systems of the abbreviations so that renaming of only the surface part E^α became possible. This, however, changes hash values, and moreover, allowes one term to occur multiple times in different surfaces E^α without being renamed.

To circumvent this later renaming and hence, the necessity to use multiple hash tables, the solution given in the appendix works differently: The function Renamings given in Figure A.1 computes all possible scopes *before the actual compilation*: The function call Renamings(S) returns a pair (L, Υ), where L is the set of locally declared variables in S and Υ is a *list of substitutions*[7]. For example, for the program given in Figure 1, we obtain $L = \{a, b, c\}$ and $\Upsilon = [\rho_1, \rho_2, \rho_3, \{\}]$ with $\rho_1 = \{(a, a_1), (b, b_1), (c, c_1)\}$, $\rho_2 = \{(b, b_2), (c, c_2)\}$, and $\rho_3 = \{(c, c_3)\}$. The number of substitutions in Υ is the maximal nesting of loops in S plus one (we have one substitution for every surface of a loop and another one for the depth).

Having computed the renamings in advance, we can *rename terms before storing them in a hash table*. To this end, all required renamings Υ are forwarded by CompileDepth to the function CompileSurface to compute a correctly renamed surface of loop bodies (see the code for translating loops).

In the functions of the appendix, 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. The function NewVar generates a new variable, and NewDef(τ) either returns the variable associated with the expression τ (if τ already appears in the hash table) or it generates a new entry in the hash table with a new variable that abbreviates τ. Finally, we use functions for manipulating lists like Cons for adding a leftmost element, Head to copy the leftmost element, and Tail to cut it off. The following theorem summarizes the invariants of our construction:

[7] A substitution is a set of pairs (x, τ) meaning that the variable x has to be replaced by the term τ.

Theorem 4.2 (Translating Quartz Statements) *Given a* Quartz *statement S, start conditions go^α and go^η, a renaming ρ of the local variables of S, the function call* CompileSurface $(\rho, go^\alpha, go^\eta, S)$ *computes a triple (I, G^α, R^α) with the following meaning (Figure A.5):*

- $I = \mathsf{inst}(S)$
- $G^\alpha = \mathsf{guardcmd}(go^\alpha, \mathsf{surface}(S))$
- $\alpha_L(\mathcal{R}_{\mathsf{cf}}(go^\eta, S)) \Leftrightarrow \bigwedge_{\tau \in R^\alpha} \tau$, *where L are the locations of S*

Given a suspension condition sp, an abortion condition kl, and renamings for all surfaces of nested loops Υ, the call CompileDepth (Υ, sp, kl, S) *yields a tuple $(C, L, \Xi, A, T, G^\eta, R^\eta)$ such that the following holds (Figures A.2-A.4):*

- *C is the set of added control variables*[8] *to eliminate nondeterministic statements (nondeterministic choice and asynchronous concurrency)*
- *L is the set of variables that are locally declared in S*
- *Ξ is a list that contains for all surfaces of loops in S a set of tuples (go^η, x, x') such that x' is the renaming of a local variable $x \in L$ that is used in the surface with start condition go^η*
- $A = \mathsf{in}(S)$
- $T = \mathsf{term}(S)$
- $G^\eta = \mathsf{guardcmd}(go^\alpha, \mathsf{depth}(S))$
- $\eta_{go^\eta}(\mathcal{R}_{\mathsf{cf}}(go^\eta, S)) \Leftrightarrow \bigwedge_{\tau \in R^\eta} \tau$

Finally, using a new start location ℓ_0, StartCompile(ℓ_0, S) *(Figure A.2) computes a tuple (C, L, Ξ, G, R) with the control variables C, the renamings with start conditions Ξ, the guarded commands G and the control flow equations R. G and R represent the data and control flow of S, respectively.*

The runtimes of StartCompile(ℓ_0, S) and CompileDepth (Υ, sp, kl, S) are of order $O(|S|^2)$. The reason for the quadratic blow up is that sequences and loops generate copies of surfaces of their substatements as shown in Figure 2 (the lines separate surface and depth of the entire statement). CompileSurface $(\rho, go^\alpha, go^\eta, S)$ runs in time $O(|S|)$.

Consider now the *translation of local declarations*: Recall that the reason for a schizophrenic local declaration is that its depth is executed together with at least one of its surfaces. This may only happen when the local declaration is contained in a loop! Hence, in the translation of loops, we rename all local

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

- $S_1; S_2 :\equiv \left(\begin{array}{l} \mathsf{surface}\,(S_1)\,; \\ \textit{if}\ \mathsf{inst}\,(S_1)\ \textit{then}\ \mathsf{surface}\,(S_2)\ \textit{end}; \\ \mathsf{depth}\,(S_1)\,; \\ \textit{if}\ \mathsf{in}\,(S_1)\ \textit{then}\ \mathsf{surface}\,(S_2)\ \textit{end}; \\ \mathsf{depth}\,(S_2) \end{array}\right)$
- $\textit{do}\ S\ \textit{while}\ \sigma :\equiv \left(\begin{array}{l} \mathsf{surface}\,(S) \\ \textit{do} \\ \quad \mathsf{depth}\,(S)\,; \\ \quad \textit{if}\ \sigma\ \textit{then}\ \mathsf{surface}\,(S)\ \textit{end} \\ \textit{while}\ \sigma \end{array}\right)$

Figure 2. Sequences and Loops Generate Copies of Surfaces

variables in the surface that is computed by CompileDepth with the first renaming in Υ. The deeper nested surfaces are renamed with the remaining substitutions. Hence, CompileDepth only forwards the renamings and transfers them to the calls to CompileSurface in the translation of loops. Note, however, that the surface copies of sequences are not renamed, since they are not responsible for schizophrenia.

Note again, that these renamings are forwarded through the compilation, so that there is no explicit renaming step (in contrast to the proceedings version of this article). The renaming finally takes place in function CompileSurface by (1) renaming all conditions σ of conditional statements and (2) by calling function RenameAction when actions are compiled. Note that only current values are thereby substituted, so that delayed actions can correctly transfer their computed values to the depth.

The renaming alone, however, is not sufficient: Additionally, we have to disable delayed actions on the local variables that would otherwise take place when the loop body terminates. Without disabling these actions, they would erroneously transfer values to the next macro step of the new scope. This 'disabling' is done in the translation of loops with the function DisableDelayedLocals that is defined as $\mathsf{DisableDelayedLocals}(L, \delta, G) := \{\mathsf{DisableDL}(L, \delta, (\gamma, \mathcal{C})) \mid (\gamma, \mathcal{C}) \in G\}$, where:

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

This completes the translation. Note that the number of possible renamings of a local variable depends on the number of surrounding loops. We only rename when loops are encountered, in contrast to [15,22], where renaming is made in the translation of local declarations. Nevertheless, 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 reincarn-

ation is possible by examining the satisfiability of the start and termination condition of the local declaration.

5 Summary

We have shown how synchronous programs of the Esterel-family can be compiled into intermediate data structures that can be used for program analysis like verification and also for code generation. In particular, we have considered the compilation of locally declared variables which is difficult for synchronous languages due to the phenomenon of schizophrenia problems. This means that different scopes of a local declaration that may exist at the same point of time have to be safely distinguished. To this end, our compilation technique makes use of sophisticated renaming steps and also disables delayed actions on local variables to avoid interference between different scopes.

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 pp. 425–454.

[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] Dijkstra, E., *Guarded commands, nondeterminacy and formal derivation of programs*, Communications of the ACM **18** (1975), pp. 453–457.

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

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

[10] 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.

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

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

[13] Lavagno, L. and E. Sentovich, *ECL: A specification environment for system-level design*, in: *International Design Automation Conference (DAC)* (1999), pp. 511–516.

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

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

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

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

[18] Schneider, K., *Proving the equivalence of microstep and macrostep semantics*, in: V. Carreño, C. Muñoz and S. Tahar, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOL)*, LNCS **2410** (2002), pp. 314–331.

[19] Schneider, K., J. Brandt and T. Schuele, *Causality analysis of synchronous programs with delayed actions*, in: *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)* (2004), pp. 179–189.

[20] Schneider, K., J. Brandt, T. Schuele and T. Tuerk, *Improving constructiveness in code generators*, in: *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005.

[21] Schneider, K., J. Brandt, T. Schuele and T. Tuerk, *Maximal causality analysis*, in: *Conference on Application of Concurrency to System Design (ACSD)* (2005), pp. 106–115.

[22] 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.

[23] Sentovich, E., *Quick conservative causality analysis*, in: *International Symposium on System Synthesis (ISSS)* (1997), pp. 2–8.

[24] Shiple, T., "Formal Analysis of Synchronous Circuits," Ph.D. thesis, University of California at Berkeley (1996).

[25] Shiple, T., G. Berry and H. Touati, *Constructive analysis of cyclic circuits*, in: *European Design and Test Conference (EDTC)* (1996), pp. 328–333.

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

[27] Tardieu, O. and R. de Simone, *Curing schizophrenia by program rewriting in Esterel*, in: *Formal Methods and Models for Codesign (MEMOCODE)* (2004), pp. 39–48.

A Implementation

```
function Merge(Υ1, Υ2)
 if Υ1 = [ ] then return Υ2
 elsif Υ2 = [ ] then return Υ1
 else
    ρ := Head(Υ1) ∪ Head(Υ2);
    Υ := Merge(Tail(Υ1), Tail(Υ2));
    return Cons(ρ, Υ)
 end
end function

function Renamings(P)
 case P of
    nothing,
    emit x, emit next(x), now σ, y := τ, next(y) := τ,
    ℓ : pause :
        return ({}, [{}])
    if σ then S1 else S2 end :
    choose S1 [] S2 end :
    S1; S2 :
    S1 ‖ S2 :
    S1 ||| S2 :
        (L1, Υ1) := Renamings(S1); (L2, Υ2) := Renamings(S2);
        return (L1 ∪ L2, Merge(Υ1, Υ2))
    do S while σ :
        (L, Υ) := Renamings(S);
        ϱ := {(x, NewVar()) | x ∈ L};
        return (L, Cons(ϱ, Υ))
    suspend S when σ : return Renamings(S)
    weak suspend S when σ : return Renamings(S)
    abort S when σ : return Renamings(S)
    weak abort S when σ : return Renamings(S)
    during S holds σ : return Renamings(S)
    local x in S end, local x : α in S end :
        (L, Υ) := Renamings(S);
        return ({x} ∪ L, Υ)
 end case
end function
```

Figure A.1. Precomputation of Required Renamings

B Generating Data Flow of Guarded Commands

Using the algorithms given in this paper, we can extract the control and data flow of every Quartz program. In particular, the function StartCompile as given in Appendix A computes a tuple (C, L, Ξ, G, R) where G and R represent the data and the control flow, respectively. The control flow R is thereby a set of equations that contains for each location ℓ of the program a transition equation of the form $\mathsf{next}(\ell) = \varphi$. These transition equations can be directly used for code generation as their right hand sides do not refer to future values. Hence, they can be evaluated in the current variable assignment to determine the locations that hold the control flow at the next point of time.

The data flow is represented by the guarded commands G that are also computed by the function StartCompile. A guarded command is a pair $(\gamma, \mathcal{C})$ where γ is called the trigger condition and $\mathcal{C}$ is an action, i.e., either a delayed/immediate emission, a delayed/immediate assignment or a constraint of the form $\mathit{now}(\sigma)$. The meaning is that whenever γ holds, we immediately execute the action $\mathcal{C}$ (in case of emissions or assignments) or demand that the condition σ holds.

In contrast to the control flow, the data flow is not directly in an executable form: potential write conflicts and/or causality cycles are further problems for code generation. For this reason, additional checks are necessary to complete the code generation. In the following, we first describe how an initial condition and a transition relation can be derived as the basis for further checks.

function StartCompile(ℓ_0, P)
 $(L, \Upsilon) :=$ Renamings(P);
 $(I, G^\alpha, R^\alpha) :=$ CompileSurface $(\{\}, \ell_0, \ell_0, P)$;
 $(C, L, \Xi, A, T, G^\eta, R^\eta) :=$ CompileDepth $(\Upsilon, \mathsf{false}, \mathsf{false}, P)$;
 return $(C, L, \Xi, G^\alpha \cup G^\eta, \{\mathsf{next}(\ell_0) = \mathsf{false}\} \cup (R^\alpha \& R^\eta))$
end function

function CompileDepth (Υ, sp, kl, P)
 case P **of**
  nothing : **return** $(\{\}, \{\}, [\,], \mathsf{false}, \mathsf{false}, \{\}, \{\})$
  ℓ : *pause* : **return** $(\{\}, \{\}, [\,], \ell, \ell, \{\}, \{\mathsf{next}(\ell) = sp \wedge \ell\})$
  emit x, *emit next*(x), *now* σ, $y := \tau$, *next*$(y) := \tau$:
   return $(\{\}, \{\}, [\,], \mathsf{false}, \mathsf{false}, \{\}, \{\})$
  choose S_1 [] S_2 *end* :
   $c :=$ NewVar();
   $(C, L, \Xi, A, T, G, R) :=$ ConditionalCompileDepth $(\Upsilon, sp, kl, c, S_1, S_2)$;
   return $(C \cup \{c\}, L, \Xi, A, T, G, R)$
  $S_1 \,|||\, S_2$:
   $c_1 :=$ NewVar(); $P_1 :=$ *suspend* S_1 *when* $\neg c_1$;
   $c_2 :=$ NewVar(); $P_2 :=$ *suspend* S_2 *when* $\neg c_2$;
   $\sigma := [\mathsf{in}\,(S_1) \wedge c_1] \vee [\mathsf{in}\,(S_2) \wedge c_2]$;
   $P :=$ *during* $P_1 \,\|\, P_2$ *holds* σ;
   $(C, L, \Xi, A, T, G, R) :=$ CompileDepth (Υ, sp, kl, P);
   return $(C \cup \{c_1, c_2\}, L, \Xi, A, T, G, R)$
  $S_1 \,\|\, S_2$: **return** ParallelCompileDepth $(\Upsilon, sp, kl, S_1, S_2)$
  $S_1; S_2$: **return** SequenceCompileDepth $(\Upsilon, sp, kl, S_1, S_2)$
  if σ *then* S_1 *else* S_2 *end* : **return** ConditionalCompileDepth $(\Upsilon, sp, kl, \sigma, S_1, S_2)$
  do S *while* σ : **return** DoWhileCompileDepth $(\Upsilon, sp, kl, \sigma, S)$
  suspend S *when* σ : **return** SuspendCompileDepth $(\Upsilon, sp, kl, \sigma, S, \mathsf{false})$
  weak suspend S *when* σ : **return** SuspendCompileDepth $(\Upsilon, sp, kl, \sigma, S, \mathsf{true})$
  abort S *when* σ : **return** AbortCompileDepth $(\Upsilon, sp, kl, \sigma, S, \mathsf{false})$
  weak abort S *when* σ : **return** AbortCompileDepth $(\Upsilon, sp, kl, \sigma, S, \mathsf{true})$
  local x *in* S *end*, *local* $x : \alpha$ *in* S *end* :
   $(C, L, \Xi, A, T, G, R) :=$ CompileDepth (Υ, sp, kl, S);
   return $(C, L \cup \{x\}, \Xi, A, T, G, R)$
  during S *holds* σ :
   $(C, L, \Xi, A, T, G, R) :=$ CompileDepth (Υ, sp, kl, S);
   return $(C, L, \Xi, A, T, G \cup \{(A, \mathit{now}\ \sigma)\}, R)$
 end case
end function

Figure A.2. Computing Depth Parts of Quartz Statements (Part I)

Second, we describe how code can be derived in form of equation systems provided that there are neither write conflicts nor causality cycles.

B.1 Constraints and Invariants

Constraints that are given by the *during* and *now* statements are normally used for verification. For code generation, they can be used to produce warnings. Assume that we have computed the constraints $(\gamma_1, \mathit{now}(\sigma_1)), \ldots, (\gamma_p, \mathit{now}(\sigma_p))$ for a given program. The meaning of these constraints is that the following formula invariantly holds:

$$\mathsf{invar}_{\mathsf{now}} :\equiv \bigwedge_{i=1}^{p} (\gamma_i \rightarrow \sigma_i)$$

For this reason, we simply *add this formula to the initial condition and to the transition relation* which has the effect that all initial states satisfy the constraints and that all states that violate this condition do not have outgoing transitions. Hence, all reachable states satisfy the constraints.

Note, however, that there are still states that violate condition $\mathsf{invar}_{\mathsf{now}}$. However, since these states are not reachable, they can be neglected. As the formula $\mathsf{invar}_{\mathsf{now}}$ approximates the reachable

function ConditionalCompileDepth $(\Upsilon, sp, kl, \sigma, S_1, S_2)$
$(C_1, L_1, \Xi_1, A_1, T_1, G_1, R_1) := \mathsf{CompileDepth}\,(\Upsilon, sp, kl, S_1)$;
$(C_2, L_2, \Xi_2, A_2, T_2, G_2, R_2) := \mathsf{CompileDepth}\,(\Upsilon, sp, kl, S_2)$;
$A := \mathsf{NewDef}(A_1 \vee A_2)$;
$T := \mathsf{NewDef}(T_1 \vee T_2)$;
$G := G_1 \cup G_2$;
$R := R_1 \cup R_2$;
return $(C_1 \cup C_2, L_1 \cup L_2, \mathsf{Merge}(\Xi_1, \Xi_2), A, T, G, R)$
end function

function SequenceCompileDepth $(\Upsilon, sp, kl, S_1, S_2)$
$(C_1, L_1, \Xi_1, A_1, T_1, G_1, R_1) := \mathsf{CompileDepth}\,(\Upsilon, sp, kl, S_1)$;
$go_2^\eta := \mathsf{NewDef}(T_1 \wedge \neg(sp \vee kl))$;
$(I_2, G_2^\alpha, R_2^\alpha) := \mathsf{CompileSurface}\,(\{\}, T_1, go_2^\eta, S_2)$;
$(C_2, L_2, \Xi_2, A_2, T_2, G_2^\eta, R_2^\eta) := \mathsf{CompileDepth}\,(\Upsilon, sp, kl, S_2)$;
$G := G_1 \cup G_2^\alpha \cup G_2^\eta$;
$R := R_1 \cup (R_2^\alpha \& R_2^\eta)$;
$A := \mathsf{NewDef}(A_1 \vee A_2)$;
$T := \mathsf{NewDef}(T_1 \wedge I_2 \vee T_2)$;
return $(C_1 \cup C_2, L_1 \cup L_2, \mathsf{Merge}(\Xi_1, \Xi_2), A, T, G, R)$
end function

function DoWhileCompileDepth $(\Upsilon, sp, kl, \sigma, S)$
$(C, L, \Xi, A, T, G^\eta, R^\eta) := \mathsf{CompileDepth}\,(\mathsf{Tail}(\Upsilon), sp, kl, S)$;
$go^\alpha := \mathsf{NewDef}(T \wedge \sigma)$;
$go^\eta := \mathsf{NewDef}(go^\alpha \wedge \neg(sp \vee kl))$;
$\rho := \mathsf{Head}(\Upsilon)$;
$\rho' := \{(go^\eta, x, x') \mid (x, x') \in \rho \wedge x \in L\}$;
$(I, G^\alpha, R^\alpha) := \mathsf{CompileSurface}\,(\rho, go^\alpha, go^\eta, S)$;
$T := \mathsf{NewDef}(T \wedge \neg\sigma)$;
$\delta := \mathsf{NewDef}(T \vee A \wedge kl)$;
$G^\eta := \mathsf{DisableDelayedLocals}(L, \delta, G^\eta)$;
return $(C, L, \mathsf{Cons}(\rho', \Xi), A, T, G^\alpha \cup G^\eta, (R^\alpha \& R^\eta))$
end function

Figure A.3. Computing Depth Parts of Quartz Statements (Part II)

states, it can be used as a formula to describe the *care set* for BDD minimization. Additionally, we could add the conjunct $\mathsf{next}(\mathsf{invar}_{\mathsf{now}})$ to the transition relation, so that all transitions are deleted that have target states that violate condition $\mathsf{invar}_{\mathsf{now}}$. However, this is not necessary.

B.2 Event Variables

Code generation for global event variables is just a special case of code generation for local event variables. The only difference is that the latter additionally has to consider the behaviors of the reincarnated variables, while global variables simply have no reincarnations. Thus, we only consider code generation for local event variables and consider global event variables as a special case where no reincarnations occur.

Assume that we have computed for a local event variable x and its reincarnations $x_1,\ldots,x_d$ the following guarded actions (note that there are no delayed actions on the reincarnated variables):

$$\begin{array}{llll} (\gamma_1, \mathit{emit}\ x), & \ldots, & (\gamma_p, \mathit{emit}\ x), & (\chi_1, \mathit{emit}\ \mathit{next}(x)),\ \ldots,\ (\chi_q, \mathit{emit}\ \mathit{next}(x)), \\ (\gamma_{1,1}, \mathit{emit}\ x_1), & \ldots, & (\gamma_{1,p_1}, \mathit{emit}\ x_1), & \\ \vdots & & \vdots & \\ (\gamma_{d,1}, \mathit{emit}\ x_d), & \ldots, & (\gamma_{d,p_d}, \mathit{emit}\ x_d) & \end{array}$$

The meaning of an event variable is that it holds at a point of time t iff either an immediate emission is executed at time t or a delayed emission has been executed at time $t-1$ (if $t > 0$ holds).

function ParallelCompileDepth $(\Upsilon, sp, kl, S_1, S_2)$
$(C_1, L_1, \Xi_1, A_1, T_1, G_1, R_1) := \mathsf{CompileDepth}\,(\Upsilon, sp, kl, S_1)$;
$(C_2, L_2, \Xi_2, A_2, T_2, G_2, R_2) := \mathsf{CompileDepth}\,(\Upsilon, sp, kl, S_2)$;
$A := \mathsf{NewDef}(A_1 \vee A_2)$;
$T := \mathsf{NewDef}(T_1 \wedge \neg A_2 \vee T_2 \wedge \neg A_1 \vee T_1 \wedge T_2)$;
$G := G_1 \cup G_2$;
$R := R_1 \cup R_2$;
return $(C_1 \cup C_2, L_1 \cup L_2, \mathsf{Merge}(\Xi_1, \Xi_2), A, T, G, R)$
end function

function SuspendCompileDepth $(\Upsilon, sp, kl, \sigma, S, wk)$
$sp_1 := \mathsf{NewDef}(sp \vee \sigma \wedge \neg kl)$;
$(C, L, \Xi, A, T_1, G, R) := \mathsf{CompileDepth}\,(\Upsilon, sp_1, kl, S)$;
$T := \mathsf{NewDef}(T_1 \wedge \neg sp_1)$;
if $\neg wk$ **then** $G := \{(\gamma \wedge \neg\sigma, \mathcal{C}) \mid (\gamma, \mathcal{C}) \in G\}$ **end**;
return (C, L, Ξ, A, T, G, R)
end function

function AbortCompileDepth $(\Upsilon, sp, kl, \sigma, S, wk)$
$kl_1 := \mathsf{NewDef}(kl \vee \sigma)$;
$(C, L, \Xi, A, T_1, G, R) := \mathsf{CompileDepth}\,(\Upsilon, sp, kl_1, S)$;
$T := \mathsf{NewDef}(T_1 \vee A \wedge kl_1)$;
if $\neg wk$ **then** $G := \{(\gamma \wedge \neg\sigma, \mathcal{C}) \mid (\gamma, \mathcal{C}) \in G\}$ **end**;
return (C, L, Ξ, A, T, G, R)
end function

function RenameAction$(\rho^\alpha, \mathcal{C})$
case $\mathcal{C}$ **of**
emit x: **return** *emit* $\rho^\alpha(x)$
emit next(x): **return** *emit next*(x)
now σ: **return** *now* $\rho^\alpha(\sigma)$
$y := \tau$: **return** $\rho^\alpha(y) := \rho^\alpha(\tau)$
next$(y) := \tau$: **return** *next*$(y) := \rho^\alpha(\tau)$
end case
end function

Figure A.4. Computing Depth Parts of Quartz Statements (Part III)

Hence, the semantics of the reincarnated variables $x_1, \ldots, x_d$ and the locally declared variable x are defined as follows:

$$\begin{aligned}
\mathsf{invar}_{x_1} &:\equiv x_1 \leftrightarrow \bigvee_{i=1}^{p_1} \gamma_{1,i} \\
&\vdots \\
\mathsf{invar}_{x_d} &:\equiv x_d \leftrightarrow \bigvee_{i=1}^{p_d} \gamma_{d,i} \\
\mathsf{init}_x &:\equiv \bigvee_{i=1}^{p} \gamma_i \\
\mathsf{trans}_x &:\equiv \mathsf{next}\,(x) \leftrightarrow \left(\bigvee_{i=1}^{q} \chi_i\right) \vee \mathsf{next}\left(\bigvee_{i=1}^{p} \gamma_i\right)
\end{aligned}$$

As there are no delayed emissions on reincarnated variables, it follows that a reincarnated event variable x_i holds iff one of the trigger conditions currently holds, which is directly implemented by the above equations. In contrast, we have to consider immediate and delayed emissions for the original local event variable. At the initial point of time, only immediate emissions have to be considered, which yields a simple equation for the initial point of time. For all points of time $t > 0$, the variable x holds iff an immediate emission takes place or a delayed emission has been executed at the previous point of time. This is implemented by equation trans_x.

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

Figure A.5. Computing Renamed Surface Parts

Note that there is no interference between the reincarnations x_i and original variable x, since there can not be any delayed actions on the reincarnations. Hence, *reincarnated event variables can be treated as independent variables.*

The above equations can not be directly used for code generation, since the transition equation trans_x refers to the next point of time. To circumvent this problem, we introduce an intermediate variable x' that holds iff a delayed emission for x has been executed at the previous point of time. Using this variable x', code generation can be done by the following equation system:

$$\begin{aligned}
\mathsf{invar}_{x_1} &:\equiv x_1 \leftrightarrow \bigvee_{i=1}^{p_1} \gamma_{1,i} \\
&\vdots \\
\mathsf{invar}_{x_d} &:\equiv x_d \leftrightarrow \bigvee_{i=1}^{p_d} \gamma_{d,i}
\end{aligned}$$

$$\begin{aligned}
\mathsf{init}_x &:\equiv x' \leftrightarrow \mathsf{false} \\
\mathsf{trans}_{x'} &:\equiv \mathsf{next}\,(x') \leftrightarrow \bigvee_{i=1}^{q} \chi_i \\
\mathsf{invar}_x &:\equiv x \leftrightarrow x' \vee \bigvee_{i=1}^{p} \gamma_i
\end{aligned}$$

There are no write conflicts on event variables, since it is not possible to 'unemit' an event variable. However, there may be causality conflicts in case the above equation system contains cyclic dependencies.

An important special case where no causality cycles can appear is the case where the program has no immediate emissions on the event variable x. In this case, there are no emissions on reincarnated variables at all, so that these variables are all invariantly false. Moreover, it is easily seen that equation invar_x becomes simply $x \leftrightarrow x'$, so that x' is no longer necessary. Thus, we obtain the simplified equation system:

$$\begin{aligned}
\mathsf{invar}_{x_1} &:\equiv x_1 \leftrightarrow \mathsf{false} \\
&\vdots \\
\mathsf{invar}_{x_d} &:\equiv x_d \leftrightarrow \mathsf{false} \\
\mathsf{init}_x &:\equiv x \leftrightarrow \mathsf{false} \\
\mathsf{trans}_x &:\equiv \mathsf{next}\,(x) \leftrightarrow \bigvee_{i=1}^{q} \chi_i
\end{aligned}$$

We may also consider the special case, where there are only immediate emissions on x. In this case, there is also no need for the introduction of x' since x' is easily seen to be invariantly false. Thus, we would obtain a simpler equation system that however may suffer from cyclic dependencies. These cyclic dependencies have to be checked by means of causality analysis [14,24,25,23,5,19,20,21]. We do not consider the details of causality analysis in this article, but note that in many cases (though not in all cases where an equivalent acyclic system exists), causality analysis can transform a cyclic equation system into an acyclic one with the cost of larger right hand side formulas.

B.3 State Variables

Similar to event variables, code generation for global (output) variables can be considered as a special case of code generation for local state variables. Hence, we only consider local state variables in the following.

In contrast to event variables, there may be an interference of reincarnated variables and the original state variable x: It may be the case that there is neither a delayed assignment to x in the surface of the local declaration nor an immediate assignment right after this point of time. In this case, x has to capture the value of the reincarnated variable x_i that corresponds with the outermost surface that has been executed. If there were no (immediate) assignments to this reincarnated variable x_i, then x is (re)initialized with the default value of the type of x.

Hence, we have to determine the 'outermost executed surface' in order to capture the value of the least recently reincarnated variable. To this end, the start condition go^η of each surface of the local declaration has been stored in the list Ξ. Using these conditions, we are able to check one after the other (in the ordering of their nestings) to determine the outermost activated surface.

To describe this formally, assume we have computed for a local state variable x and its d reincarnations $x_1,\ldots,x_d$ the following guarded actions (note again that there are no delayed actions on the reincarnations), where x_1 is assumed to be the outermost reincarnation of x:

$$\begin{array}{llll}
(\gamma_1, x := \tau_1), & \ldots, & (\gamma_p, x := \tau_p), (\chi_1, next(x) := \pi_1), \ldots, (\chi_q, next(x) := \pi_q) \\
(\gamma_{1,1}, x_1 := \tau_{1,1}), & \ldots, & (\gamma_{1,p_1}, x_1 := \tau_{1,p_1}) \\
\vdots & & \vdots \\
(\gamma_{d,1}, x_d := \tau_{d,1}), & \ldots, & (\gamma_{d,p_d}, x_d := \tau_{d,p_d})
\end{array}$$

Furthermore, assume that go_i is the start condition of the loop body that generated the surface where the local variable x has been reincarnated to x_i (hence, $\bigvee_{j=1}^{p_i} \gamma_{i,j} \rightarrow go_i$ must be valid, but the converse may not hold).

We first consider the construction of an initial condition and a transition relation that can be used for checking problems like write conflicts and causality cycles. To this end, we first compute for each reincarnated variable x_i the following invariant:

$$\mathsf{invar}_{x_i} :\equiv \left(\left(\bigwedge_{j=1}^{p_i} (\gamma_{i,j} \rightarrow x_i = \tau_{i,j}) \right) \wedge \left(\left(\bigwedge_{j=1}^{p_i} \neg\gamma_{i,j} \right) \rightarrow x_i = \tau_0 \right) \right)$$

This invariant simply states that whenever a trigger condition $\gamma_{i,j}$ holds, then the immediate assignment $x_i := \tau_{i,j}$ is executed so that the equation $x_i = \tau_{i,j}$ must hold. If more than one trigger condition $\gamma_{i,j}$ holds, then there may be a write conflict since there are two assignments to x_i at the same point of time. We may further check if these multiple assignments are performed with the same value or not, thus obtaining more precise, but more expensive semantic checks.

If no trigger condition $\gamma_{i,j}$ holds, then we demand that x_i equals to the default value τ_0 of the type of x. In principle, this would only be necessary in case that the local declaration is started, but it is safe to assign the default value whenever there is no other assignment.

A similar formula is used for the initial condition of the variable x, which emphasizes that the semantics of reincarnated variables corresponds with an initialization:

$$\mathsf{init}_x :\equiv \left(\left(\bigwedge_{j=1}^{p} (\gamma_j \rightarrow x = \tau_j) \right) \wedge \left(\left(\bigwedge_{j=1}^{p} \neg\gamma_j \right) \rightarrow x = \tau_0 \right) \right)$$

The transition relation of x is more difficult. First of all, $x = \tau_j$ must hold if the trigger condition γ_j holds. Second, if the trigger condition χ_j of a delayed assignment $\mathit{next}(x) := \pi_j$ holds, then x must have the value π_j at the next point of time (note that π_j is evaluated with the current variables to determine the value of x for the next point of time). If neither a trigger condition γ_j of an immediate assignment nor a trigger condition of a delayed assignment χ_j holds, then x has to maintain its previous value. There is one difficult problem that has to be faced for local state variables: It may be the case that the local declaration of x has been entered at the previous point of time, so that one of the reincarnated variable's values has to be stored. In this case, we have to determine the outermost active surface to refer to the correct reincarnation. To this end, we use a case construct to check the start conditions go_i one after the other. Note that if go_i is the first one that holds, then all go_{i+j} do also hold, which means that go_i refers to the outermost surface whose reincarnated variable has to be referred to. Finally, if no start condition go_i holds, then the control flow moved from somewhere inside the scope of x to somewhere inside the scope of x without leaving the scope in between. Hence, we simply store the previous value of x, which is the meaning of a *state* variable:

$$\mathsf{trans}_x :\equiv \left(\begin{array}{l} \left(\bigwedge_{j=1}^{p} (\gamma_j \rightarrow x = \tau_j) \right) \wedge \\ \left(\bigwedge_{j=1}^{q} (\chi_j \rightarrow \mathsf{next}(x) = \pi_j) \right) \wedge \\ \left(\mathsf{next}\left(\bigwedge_{j=1}^{p} \neg\gamma_j \right) \wedge \left(\bigwedge_{j=1}^{q} \neg\chi_j \right) \rightarrow \mathsf{next}(x) = \left(\begin{array}{l} \textbf{case} \\ \quad go_1 : x_1; \\ \quad \vdots \\ \quad go_d : x_d; \\ \textbf{else } x \\ \textbf{end case} \end{array} \right) \right) \end{array} \right)$$

Again, it is possible that write conflicts can appear. Note that it is no problem if several surfaces are executed and if the local variable is assigned different values in each of these surfaces. Write conflicts appear, however, if different values are assigned at the same point of time, regardless of whether the assignment refers to a reincarnated variable x_i or to the original variable x. Of course, x may additionally suffer from a write conflict if a delayed assignment is followed by an immediate assignment with a different value.

In case a write conflict appears, the transition relation degenerates to false, which means that states where a write conflict appears do not have outgoing transitions. Hence, we can simply check for reachable states that have no outgoing transitions in order to check the presence of write conflicts.

Similar to event variables, the transition relation is not constructive in the sense that we can use it directly for code generation. In case that there are neither causality cycles nor write conflicts, we can derive simple equation systems for code generation. We briefly consider the definition of such equation systems for the abovely considered local state variable x and its reincarnations $x_1,\ldots,x_d$. To this end, we will use case-constructs to check the trigger conditions one after the other in order to find one that holds to determine a value that has to be assigned.

For each reincarnation x_i, we use the following equation with such a case-construct, where τ_0 is again the default value associated with the type of x:

$$\mathsf{invar}_{x_i} :\equiv x_i = \left(\begin{array}{l}\textbf{case}\\ \quad \gamma_{i,1} : \tau_{i,1};\\ \quad\quad \vdots\\ \quad \gamma_{i,p_i} : \tau_{i,p_i};\\ \textbf{else } \tau_0\\ \textbf{end case}\end{array}\right)$$

The above equation guarantees that the assignments take place whenever the corresponding guard $\gamma_{i,j}$ holds (note that we excluded write conflicts, so that at most one of the guards can hold). In case that none of the guards holds, we assign the default value to x_i, so that the initialization is guaranteed whenever the local declaration is entered without executing immediate assignments. Again, this is more restrictive than necessary. However, this is reasonable to generate deterministic code.

Similar to the transition relation, we could construct an initial equation for x that would resemble the above invariant of the reincarnated variables x_i. However, the transition equation for x is more difficult, since a straightforward construction would refer to values of the trigger conditions of immediate assignments at the next point of time. To circumvent this, we use the same trick as for event variables, i.e., we introduce an auxiliary variable x' to capture a delayed assignment at the previous point of time. Before considering the equations for x', let us define the invariant of x:

$$\mathsf{invar}_{x} :\equiv x = \left(\begin{array}{l}\textbf{case}\\ \quad \gamma_{1} : \tau_{1};\\ \quad\quad \vdots\\ \quad \gamma_{p} : \tau_{p};\\ \textbf{else } x'\\ \textbf{end case}\end{array}\right)$$

To establish the correspondence with the reincarnated variables, we demand that the initial value of x' will be the default value τ_0 of x's type. This is reasonable, since the precise meaning of x' is as follows: If $next(x) := \pi_j$ is executed at a point of time t, then we evaluate π_j at time t and assign x' this value at the next point of time. As this can not happen before the initial point of time, we start with the initial value τ_0. The transition equation is more complicated:

$$\mathsf{init}_{x'} :\equiv x' = \tau_0$$

$$\mathsf{trans}_{x'} :\equiv \mathsf{next}\,(x') = \left(\begin{array}{l}\textbf{case}\\ \quad \chi_1 : \pi_1;\\ \quad\quad \vdots\\ \quad \chi_q : \pi_q;\\ \quad go_1 : x_1;\\ \quad\quad \vdots\\ \quad go_d : x_d;\\ \textbf{else } x\\ \textbf{end case}\end{array}\right)$$

The transition equation of x' checks if one of the delayed assignments is triggered. If so, its value has to be transferred via x' to x after the delay of one unit of time (note that we assume that no write conflicts appear and therefore only one of the guards can hold). If none of the delayed assignments is triggered, then the old value of x has to be stored. However, if the scope of the local declaration should have been re-entered, then we have to refer to the outermost activated surface as already explained for the construction of the transition relation. Hence, we add further cases as we did for the construction of the transition relation.

Finally, consider the case, where only delayed assignments occur in the given program. In this case, all reincarnated variables are equal to the default value τ_0, and x equals to x'. Thus, we can directly use x instead of x' and obtain the following acyclic equations:

$$
\begin{aligned}
\mathsf{invar}_{x_1} &:\equiv x_1 = \tau_0 \\
&\vdots \\
\mathsf{invar}_{x_d} &:\equiv x_d = \tau_0 \\
\mathsf{init}_x &:\equiv x = \tau_0 \\
\mathsf{trans}_x &:\equiv \mathsf{next}(x) = \left(\begin{array}{l}
\textbf{case} \\
\quad \chi_1 : \pi_1; \\
\quad\quad \vdots \\
\quad \chi_q : \pi_q; \\
\quad \bigvee_{j=1}^{d} go_j : \tau_0; \\
\textbf{else } x \\
\textbf{end case}
\end{array}\right)
\end{aligned}
$$