A New Method for Compiling Schizophrenic Synchronous Programs

K. Schneider and M. Wenz
University of Karlsruhe, Institute for Computer Design & Fault Tolerance,
Klaus.Schneider@informatik.uni-karlsruhe.de,mwenz@ira.uka.de
http://goethe.ira.uka.de/∼schneider

ABSTRACT

Synchronous programming languages have proved to be advantageous for designing software and hardware for embedded systems. Despite their clear semantics, their compilation is remarkably difficult: In particular, one has to take care of potential schizophrenia problems. Although these problems are correctly translated with existing compilers, there is still a need for clean algorithms. In this paper, we present the first solution to eliminate schizophrenia problems by program transformations. These transformations are used for compilation, but also for increasing the readability of programs.

Keywords

Synchronous Languages, Reactive Systems, Code Generation.

1. INTRODUCTION

Synchronous languages like Esterel [1 3] or variants thereof [6 12 11 16] lend themselves well for the design of software and hardware for reactive real time systems, and are therefore already used in many industrial applications [7]. The basic paradigm of these languages is the *perfect synchrony*, which follows from the fact that most of the statements are executed as 'microsteps' in zero time. Consumption of time must be explicitly programmed with special statements like Esterel's **pause** statement: The execution of a **pause** statement consumes one logical unit of time, and therefore separates different macrosteps from each other. As the **pause** statement is the only (basic) statement that consumes time, all threads run in lockstep: they execute the code, (i.e. the microsteps) between two **pause** statements in zero time, and automatically synchronize at their next **pause** statements.

To understand the data flow of a synchronous program, it is important to know that any variable, and hence any data expression, must have a uniquely determined value in each macrostep. If a variable's value is changed by a microstep of a macrostep, then the new value is immediately seen in the entire macrostep, i.e. in all its microsteps. For this reason, the actual order in which the microsteps of a macrostep are executed is irrelevant.

The abstraction to macrosteps makes synchronous programming so attractive. It is not only an ideal programmer's model, it additionally allows the direct translation of programs into synchronous hardware circuits, where macrosteps directly correspond to clock cycles. As the same translation is also used for software generation, many optimizations known for hardware circuits can be used to optimize software as well [2 8]. For this reason, synchronous programs are a good basis for HW/SW codesign [14].

However, the abstraction to macrosteps is not for free: Causality cycles and schizophrenic statements are the two major problems that must be solved by a compiler. *Causality cycles* arise when the value of a variable depends on itself, which corresponds to combinatorial loops in hardware circuits. Algorithms for causality analysis, that check if such cycles yield stable values, have been considered in many areas, and efficient algorithms are already available [13 10 5 17 4 2].

So far, there are less algorithms to solve *schizophrenia problems*. These problems occur if a statement is executed several times at the same point of time. This may only happen if a loop body terminates and is entered within the same macrostep. If the scope of a local variable is thereby left, and a new scope of the same variable is created, the compiler must carefully distinguish between the different incarnations that exist at the same time. This yields a difficult problem for hardware synthesis [2], since outputs of gates and registers must have a uniquely determined value at each point of time. As the software generation is also based on the circuit synthesis, the same problems appear also there.

Note that schizophrenia problems are not particular problems of Esterel. They must necessarily arise in all synchronous programming languages that provide local declarations as microsteps. Although it is reported in [2] that schizophrenia problems are rare, they still must be correctly handled by a compiler. As this is remarkably difficult, some languages like certain Statechart variants do not support local declarations at all, and thus support modular programming only in a limited way.

Simple solutions of schizophrenia problems (like duplication of loop bodies) generate unnecessarily large code. This is not acceptable for embedded systems, where resources are still rare. Thus, methods to compile programs with schizophrenia problems into small target code are still of high interest. Furthermore, the compilation should be based on 'simple' transformations that allow an efficient verification, if the correctness of a program has to be verified.

The Esterel compiler [3 7] is already able to solve all kinds of schizophrenia problems. Beneath schizophrenic local declarations, also schizophrenic parallel statements have to be considered there. The solution given in [2] considers for each schizophrenic statement a couple of copies according to its 'incarnation level'. This duplication of code segments is necessary to distinguish between different incarnations, and can not be circumvented. The compile

CASES'01, November 16-17, 2001, Atlanta, Georgia, USA.

time and the code generated by the Esterel compiler is also very good. On the other hand, the procedure described in [2] is quite complicated, and therefore it is hard to extend it with optimizations, or to check its correctness, e.g. with a theorem prover.

In this paper, we present a different solution to solve schizophrenia problems, where only one copy of the local variable is necessary, regardless of its incarnation level. Our solution is based on simple program transformations that allow us to transform potentially schizophrenic programs into equivalent ones without schizophrenia problems. We have proved the correctness of these transformations with the theorem prover HOL [9]. By experimental results, we moreover show that our compiler compares well with the sophisticated Esterel compiler. In fact, there are examples where our compiler generates significantly smaller code (section 5) without further optimizations.

The paper is organized as follows: in the next section, we define the syntax and semantics of our Esterel variant Quartz [15 16]. In Section 3, we then consider the two kinds of schizophrenia problems that arise in Quartz, namely schizophrenic abortion statements and schizophrenic local declarations, and present our program transformations for eliminating schizophrenic statements. Detailed algorithms for a transformation into equivalent equation systems are given in section 4, together with a complexity analysis. We then conclude with some preliminary experimental results.

2. SYNTAX AND SEMANTICS

Quartz [15 16] is a variant of Esterel [1 3 7] that extends Esterel by delayed assignments and emissions, asynchronous concurrency, nondeterministic choice, and inline assertions. Asynchronous concurrency is important to model distributed systems, or to allow the compiler to schedule the threads in an optimal way. The same holds for nondeterministic choice. Delayed assignments and emissions are often convenient, since they follow the traditional sequential programming style and therefore allow simpler translations from conventional programming languages like C or hardware description languages like VHDL. In the following, we briefly describe the syntax and semantics of Quartz. For more details, the reader is referred to [16 15] or to the Esterel primer, which is an excellent introduction to synchronous programming [3].

2.1 Syntax and Informal Semantics

Time is modeled by the natural numbers $\mathbb{N}$, so that the semantics of a data type expression is a function of type $\mathbb{N} \to \alpha$ for some type α. Quartz distinguishes between two kinds of variables, namely *event variables* and *state variables*. The semantics of an event variable is a function of type $\mathbb{N} \to \mathbb{B}$, while the semantics of a state variable may have the more general type $\mathbb{N} \to \alpha$. The main difference between event and state variables is however the data flow: the value of a state variable y is 'sticky', i.e. if no data operation has been applied to y, then its value does not change. On the other hand, the value of an event variable x is not sticky: its value is reset to 0 (we denote Boolean values as 1 and 0) in the next macrostep, if it is not explicitly made 1 there. Hence, the value of an event variable is 1 at a point of time if and only if there is a thread that emits this variable at this point of time (i.e. a corresponding event).

Event variables are made present with the **emit** statement, while state variables are manipulated with assignments ($:=$). Of course, any event or state variable may also be an input, so that the values are determined by the environment only. **emit** statements and assignments are all data manipulating statements. The remaining basic statements of Quartz are given below:

DEFINITION 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, and σ is a Boolean expression:*

- **nothing** *(empty statement)*
- **emit** x *and* **emit delayed** x *(emissions)*
- $y := \tau$ *and* $y :=$ **delayed** τ *(assignments)*
- 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 parallel composition)*
- $S_1 \parallel\!\!\parallel S_2$ *(asynchronous parallel composition)*
- **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 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 we say that the execution of S takes time, or simply that S *consumes time*.

Let us now discuss the above basic statements: **nothing** simply does nothing, i.e., it neither consumes time, nor does it affect any data values. Executing **emit** x makes the event variable x present for the current macrostep, i.e., the value of x at that point of time is 1. Executing an assignment $y := \tau$ means that y and τ have the same values in the current macrostep. The variants **emit delayed** x and $y :=$ **delayed** τ are similarly defined as **emit** x and $y := \tau$, respectively, but with a delay of one macrostep. In the latter statement, τ is evaluated at the current point of time, and its value is passed to y at the next point of time. We emphasize that none of these statements consumes time, although the delayed versions affect values of variables at the next point of time.

There is only one basic statement that consumes time, namely the **pause** statement. It does not affect any data values. We endow **pause** statements with unique location variables ℓ that we will use as state variables to encode the control flow automaton.

if σ **then** S_1 **else** S_2 **end** is a conditional statement: depending on the value of σ in the current macrostep, either S_1 or S_2 is immediately executed. $S_1; S_2$ is the sequential execution of S_1 and S_2, i.e., we first enter S_1 and execute it. If S_1 never terminates, then S_2 is never executed at all. If, on the other hand, S_1 terminates, we immediately start S_2 and proceed with its execution.

$S_1 \parallel S_2$ denotes the synchronous parallel execution of S_1 and S_2: If $S_1 \parallel S_2$ is entered, we enter both S_1 and S_2, and proceed with the execution of both statements. As long as both S_1 and S_2 are active, both threads are executed in lockstep. If S_1 terminates, but S_2 does not, then $S_1 \parallel S_2$ behaves further as S_2 does (and vice versa). If finally S_2 terminates, then $S_1 \parallel S_2$ terminates.

Beneath the synchronous parallel execution, Quartz additionally offers asynchronous parallel execution $S_1 \parallel\!\!\parallel S_2$. The difference is that one of the threads may execute more than one macrostep while the other one executes a single one or even none. One may argue that the presence of asynchronous parallel execution contradicts

the definition of a synchronous language. However, it is not too difficult to replace $S_1 \parallel\!\!\mid S_2$ by standard Esterel statements using additional inputs [16] (that are called control variables). Another Quartz statement that does not belong to Esterel is the nondeterministic choice: **choose** $S_1 \,[]\, S_2$ **end** will nondeterministically execute either S_1 or S_2. Again, using additional input (control) variables, nondeterministic choice can be reduced to other statements [16], so that we neither consider nondeterministic choice nor asynchronous concurrency in the following.

do S **while** σ implements iteration: if this statement is entered, S is executed until it terminates. If then σ holds, S is once more executed, otherwise the loop terminates. It is required that for any input, the loop body S must not be instantaneous.

(weak) suspend S **when** σ implements process suspension: S is entered when the execution of this statement starts (regardless of the current value of σ). For the following points of time, however, the execution of S only proceeds if σ evaluates to 0, otherwise its execution is 'frozen' until σ releases the further execution. Beneath suspension, abortion of processes is an important means for the process management. This is realized with the **abort** statements: **abort** S **when** σ immediately enters S at starting time (regardless of the current value of σ). Then, S is executed as long as σ is 0. If σ becomes 1 during the execution of S, then S is immediately aborted. The 'weak' variants of suspension and abortion differ on the data manipulations at suspension or abortion time: While the strong variants ignore *all* data manipulations at abortion or suspension time, *all of them* are performed by the weak variants. There are also **immediate** variants of suspension and abortion that consider the condition σ additionally at starting time. These can be easily defined in terms of the other variants [16].

The statements **local** x **in** S **end** and **local** $y : \alpha$ **in** S **end** are used to define local event and local state variables, respectively. Their meaning is that they behave like S, but the scope of the variable x or y is limited to S. This means that the local variable is not seen outside the **local** statement. Without loss of generality, we assume in the following that there is no shadowing of variables, i.e., that all local variable names are unique and also different from input and output variables.

Quartz allows us to demand assertions that must hold when the control flow reaches certain locations: **now** σ demands that σ must hold in the current macrostep. **during** S **holds** σ behaves like S, but additionally demands that whenever the control flow is inside S, then σ must hold. There is no further execution if the condition σ does not hold; the behavior is not defined in this case.

Similar to Esterel, Quartz allows us to define modules so that systems can be hierarchically organized. Existing modules can be called in other modules via the **run** statement which textually replaces the module body. In general, a Quartz module is of the following form:

module M
  input $a_1, \ldots, a_n, b_1 : \alpha_1, \ldots, b_m : \alpha_m$;
  output $x_1, \ldots, x_p, y_1 : \beta_1, \ldots, y_q : \beta_q$;
  S
end module M

The above module with the name M therefore determines an interface in that it declares the input and output variables of the module. In case of state variables, it also specifies their types (α_j and β_j).

2.2 Formal Semantics

The semantics of Quartz and Esterel can be defined in several ways. In particular, Berry has worked out a semantics based on SOS transition rules (describing microsteps), and a direct translation into hardware circuits [2]. Recently, we have defined the control flow of a statement S by the following control flow predicates [16] $\mathsf{in}(S)$, $\mathsf{inst}(S)$, $\mathsf{enter}(S)$, and $\mathsf{term}(S)$, and the data flow by the set of guarded commands $\mathsf{gcmd}(\varphi, S)$. The equivalence to a simplified hardware synthesis [15] has been proved in [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 at this point of time, the control flow is 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.
- $\mathsf{term}(S)$ describes all conditions where the control flow is currently somewhere inside S 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 entered at the same time, for example, 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.
- $\mathsf{gcmd}(\varphi, S)$ is a set of pairs of the form $(\gamma, \mathcal{C})$, where $\mathcal{C}$ is a data manipulating statement, i.e., either an emission or an assignment. The meaning of $(\gamma, \mathcal{C})$ is that $\mathcal{C}$ is immediately executed whenever the guard γ holds.

Note that the above control flow predicates depend on time. Detailed definitions of the above predicates and the set of guarded commands are given in [16]. For example, for the body statement of module *AbortReincarnation* given in Figure 1, we obtain the following results ($\mathsf{X}\varphi$ means that φ holds at the next point of time, and st is the start signal):

- $\mathsf{in}(S) \equiv \ell$
- $\mathsf{inst}(S) \equiv 0$
- $\mathsf{enter}(S) \equiv \mathsf{X}\ell$
- $\mathsf{term}(S) \equiv 0$
- $\mathsf{move}(S) \equiv \ell \wedge \mathsf{X}\ell$
- $\mathsf{gcmd}(st, S) \equiv \{(st \vee \ell) \wedge \neg(\ell \wedge i), \textbf{emit } a), (\ell \wedge \neg i, \textbf{emit } b)\}$

Using the above predicates, one can easily define the control and the data flow of a program [16]. For module *AbortReincarnation*, we obtain the following initial conditions $\mathcal{I}_{\mathsf{cf}}$ and $\mathcal{I}_{\mathsf{df}}$, and the following transition relations $\mathcal{R}_{\mathsf{cf}}$ and $\mathcal{R}_{\mathsf{df}}$ of the control and data flow, respectively:

- $\mathcal{I}_{\mathsf{cf}} = \neg\ell$
- $\mathcal{R}_{\mathsf{cf}} = (\mathsf{X}\ell \leftrightarrow st \vee \ell)$
- $\mathcal{I}_{\mathsf{df}} = \mathcal{R}_{\mathsf{df}} = ([a = (st \vee \ell) \wedge \neg(\ell \wedge i)] \wedge [b = \ell \wedge \neg i])$

3. SCHIZOPHRENIA PROBLEMS

Simply combining the control and data flow, would not reflect the intended semantics in all cases. Subtle problems may arise when strong abortion and local declarations are nested within a loop statement. The problem is that loop bodies may be executed more than once since the loop's body can be terminated and entered at the same point of time. For this reason, these problems are called *schizophrenia or reincarnation problems* ([2], chapter 12).

For example, consider a strong abortion statement **abort** S **when** σ. If the abortion condition σ holds, and the control flow is currently inside S, no data manipulation of S should take place, since we have a strong abortion. For this reason, the formula $\mathsf{in}(S) \rightarrow \neg\sigma$ has been added as a conjunct to all guards of body statements

of strong abortion statements with condition σ (cf. [16] or section 4). However, if the strong abortion statement is nested within a loop, the loop immediately enters a new incarnation of the strong abortion statement. The problem is now that also the data manipulations that should take place when entering the new incarnation are suppressed. Hence, the second incarnation erroneously behaves like an immediate abortion statement.

```
module AbortReincarnation
  input i;
  output a, b;
    do
      abort
        emit a;
        ℓ : pause;
        emit b
      when i
    while 1
end
```

Figure 1: A schizophrenic abort statement

This erroneous behavior can be demonstrated with the module given in Figure 1. We expect the module to always emit the signal a after starting time, regardless of the input i, i.e., we expect the behavior of the finite state machine given in the upper half of Figure 2. The control and data flow as computed in [16] would however yield the behavior of the lower part of Figure 2 (consider state ℓ when i holds).

The problem is that the control flow leaves and enters the abort statement at the same time, so that two incarnations are concurrently exist. In the old incarnation, we have to suppress all data manipulations due to the strong abortion, but in the new incarnation all data manipulations must be performed (since we have not an immediate abort statement). Therefore, we must distinguish between the old and the new incarnation of the statement. As both incarnations exist at the same point of time, a distinction is not possible with predicates that consider macro steps only, since old and new incarnations refer to different microsteps of the same macro step. For this reason, there is no way to define the entire semantics with macro step predicates like $\mathsf{in}(S)$, $\mathsf{inst}(S)$, $\mathsf{enter}(S)$, $\mathsf{term}(S)$, $\mathsf{move}(S)$, *or any other macro step predicates.*

3.1 Surface and Depth of a Statement

To solve schizophrenia problems, we need to consider microsteps: When a statement is entered, only a part of the statement can be executed in zero time, and this part is the one whose execution may overlap with other parts of the same statement. In the following, we will call this part the *surface* and denote it as $\mathsf{surface}(S)$. The 'remaining' part of S is called the *depth* and will be written as $\mathsf{depth}(S)$.

The names 'surface' and 'depth' are borrowed from [2], section 12.5. In contrast to [2], we define $\mathsf{surface}(S)$ and $\mathsf{depth}(S)$ as

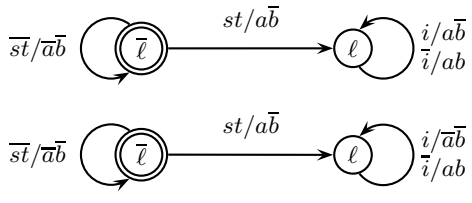

Figure 2: Intended (upper half) and erroneous (lower half) behavior of module *AbortReincarnation*

statements that are derived from S, whereas in [2], surface and depth are defined to be parts of the circuit generated from S. Our formal definition of $\mathsf{surface}(S)$ is as follows:

DEFINITION 2 (SURFACE). *For any basic* Quartz *statement* S*, we recursively define the* Quartz *statement* $\mathsf{surface}(S)$ *with the same assumptions as given in definition 1:*

- $\mathsf{surface}$ (**nothing**) $:\equiv$ **nothing**
- $\mathsf{surface}$ (**emit** x) $:\equiv$ **emit** x
- $\mathsf{surface}$ (**emit delayed** x) $:\equiv$ **emit delayed** x
- $\mathsf{surface}\,(y := \tau) :\equiv y := \tau$
- $\mathsf{surface}\,(y :=$ **delayed** $\tau) :\equiv y :=$ **delayed** τ
- $\mathsf{surface}\,(\ell :$ **pause**$) :\equiv$ **nothing**
- $\mathsf{surface}$ (**if** σ **then** S_1 **else** S_2 **end**)
  $:\equiv$ **if** σ **then** $\mathsf{surface}(S_1)$ **else** $\mathsf{surface}(S_2)$ **end**
- $\mathsf{surface}(S_1; S_2)$
  $:\equiv$ $\mathsf{surface}(S_1)$;
  if $\mathsf{inst}(S_1)$ **then** $\mathsf{surface}(S_2)$ **else nothing end**
- $\mathsf{surface}(S_1 \parallel S_2) :\equiv \mathsf{surface}(S_1) \parallel \mathsf{surface}(S_2)$
- $\mathsf{surface}$ (**do** S **while** σ) $:\equiv \mathsf{surface}(S)$
- $\mathsf{surface}$ (**suspend** S **when** σ)
  $:\equiv \mathsf{surface}$ (**weak suspend** S **when** σ)
  $:\equiv \mathsf{surface}(S)$
- $\mathsf{surface}$ (**abort** S **when** σ)
  $:\equiv \mathsf{surface}$ (**weak abort** S **when** σ)
  $:\equiv \mathsf{surface}(S)$
- $\mathsf{surface}$ (**local** x **in** S **end**) $:\equiv \mathsf{surface}(S)$
- $\mathsf{surface}$ (**local** $y : \alpha$ **in** S **end**) $:\equiv \mathsf{surface}(S)$
- $\mathsf{surface}$ (**now** σ) $:\equiv$ **now** σ
- $\mathsf{surface}$ (**during** S **holds** σ) $:\equiv \mathsf{surface}(S)$

It remains to define $\mathsf{depth}(S)$. Intuitively, $\mathsf{depth}(S)$ must not manipulate data values when it is entered, since these manipulations are completely contained in the surface part $\mathsf{surface}(S)$. However, the remaining data flow, and the entire control flow of S and $\mathsf{depth}(S)$ should be the same. A possible definition of $\mathsf{depth}(S)$ that satisfies these criteria is as follows:

DEFINITION 3 (DEPTH). *For any basic* Quartz *statement* S*, we recursively define the* Quartz *statement* $\mathsf{depth}(S)$ *with the same assumptions as given in definition 1:*

- depth (**nothing**) $:\equiv$ **nothing**
- depth (**emit** x) $:\equiv$ **nothing**
- depth (**emit delayed** x) $:\equiv$ **nothing**
- $\mathsf{depth}\,(y := \tau) :\equiv$ **nothing**
- $\mathsf{depth}\,(y :=$ **delayed** $\tau) :\equiv$ **nothing**
- $\mathsf{depth}\,(\ell :$ **pause**$) :\equiv \ell :$ **pause**
- depth (**if** σ **then** S_1 **else** S_2 **end**)
  $:\equiv$ **if** σ **then** $\mathsf{depth}(S_1)$ **else** $\mathsf{depth}(S_2)$ **end**
- $\mathsf{depth}(S_1; S_2)$
  $:\equiv \left(\begin{array}{l} \mathsf{depth}(S_1)\,; \\ \textbf{if } \mathsf{in}(S_1) \textbf{ then } \mathsf{surface}(S_2) \textbf{ end}; \\ \mathsf{depth}(S_2) \end{array}\right)$
- $\mathsf{depth}(S_1 \parallel S_2) :\equiv \mathsf{depth}(S_1) \parallel \mathsf{depth}(S_2)$
- depth (**do** S **while** σ)
  $:\equiv \left(\begin{array}{l} \textbf{do} \\ \quad \mathsf{depth}(S)\,; \\ \quad \textbf{if } \sigma \textbf{ then } \mathsf{surface}(S) \textbf{ end} \\ \textbf{while } \sigma \end{array}\right)$
- depth (**suspend** S **when** σ)
  $:\equiv$ **suspend** $\mathsf{depth}(S)$ **when** σ

- depth (**weak suspend** S **when** σ) $:\equiv$ **weak suspend** $\mathsf{depth}\,(S)$ **when** σ
- depth (**abort** S **when** σ) $:\equiv$ **abort** $\mathsf{depth}\,(S)$ **when** σ
- depth (**weak abort** S **when** σ) $:\equiv$ **weak abort** $\mathsf{depth}\,(S)$ **when** σ
- depth (**local** x **in** S **end**) $:\equiv \mathsf{depth}\,(S)$
- depth (**local** $y : \alpha$ **in** S **end**) $:\equiv \mathsf{depth}\,(S)$
- depth (**now** σ) $:\equiv$ **nothing**
- depth (**during** S **holds** σ) $:\equiv$ **during** $\mathsf{depth}\,(S)$ **holds** σ

The crucial point is that the control flows of $\mathsf{depth}\,(S)$ and S are the same. This is not trivially given, and required some special constructions in the definition of the depth for sequences and loops to avoid the introduction of new **pause** statements. Using the HOL system [9], we have proved the following theorem:

THEOREM 1 (CONTROL FLOW OF SURFACE AND DEPTH). *For any* Quartz *statement S, the following facts hold:*

- $\mathsf{surface}\,(S)$ *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*

To be more precise, we have even proved that the control flow predicates of S, $\mathsf{depth}\,(S)$ and $\mathsf{surface}\,(S)\,;\mathsf{depth}\,(S)$ are all equivalent. One might expect that also the data flow is retained. However, this is not always the case. To see this, reconsider the body statement S of module *AbortReincarnation* in Figure 1. We compute the following (with additional Boolean simplifications):

- $\mathsf{surface}\,(S) =$ **emit** a
- $\mathsf{depth}\,(S) =$
  do
    abort
      ℓ : **pause**;
      emit b
    when i;
    emit a
  while 1
- $\mathsf{gcmd}\,(st, \mathsf{surface}\,(S)) = \{(st,$ **emit** $a)\}$
- $\mathsf{gcmd}\,(st, \mathsf{depth}\,(S)) = \{(\ell,$ **emit** $a), (\ell \wedge \neg i,$ **emit** $b)\}$

Hence, $\mathsf{surface}\,(S)$; $\mathsf{depth}\,(S)$ and S do not have the same data flow: the obtained data flows are given in the upper and lower part of Figure 2. This difference can however only occur due to schizophrenic abortion statements. For this reason, we should restrict our consideration for a moment to statements where all strong abortion substatements have an empty surface, i.e., a surface with no guarded commands. Then, we have the desired equivalence of the data flows of S and $\mathsf{surface}\,(S)$; $\mathsf{depth}\,(S)$ that we have also proved with HOL:

THEOREM 2 (DATA FLOW OF SURFACE AND DEPTH). *For any* Quartz *statement S such that for any substatement of the form* **abort** P **when** σ *of S the set* $\mathsf{gcmd}\,(\varphi, \mathsf{surface}\,(P))$ *is empty for any precondition φ, the data flows of* $\mathsf{surface}\,(S)\,;\mathsf{depth}\,(S)$ *and S are the same.*

Note, however, that the sets $\mathsf{gcmd}\,(\varphi, \mathsf{surface}\,(S)\,;\mathsf{depth}\,(S))$ and $\mathsf{gcmd}\,(\varphi, S)$ need not be the same. However, under the given assumption, both sets define the same data flow as stated in the above theorem. For this reason, we have to apply the following transformation on every strong abortion substatement to eliminate potential schizophrenic abortion statements:

abort S **when** $\sigma \equiv \mathsf{surface}\,(S)\,;$**abort** $\mathsf{depth}\,(S)$ **when** σ

module *LocalReincarnation* :
output *xOn*, *xOff*;
  do
    local x **in**
      if x **then emit** *xOn* **else emit** *xOff* **end**;
      ℓ : **pause**;
      emit x;
      if x **then emit** *xOn* **else emit** *xOff* **end**
    end local
  while 1
end module

Figure 3: Reincarnation of local declarations.

The thereby obtained statement obviously fulfills the requirement of the above theorem. Using a microstep based semantics as given in [2], we can prove that the above two statements are equivalent. Alternatively, we prefer to use the above transformation to *define* the semantics of abortion statements, thus changing the semantics given in [16] to agree with Esterel's semantics.

3.2 Schizophrenic Local Declarations

In the previous subsection, we have shown how schizophrenic abortion statements are eliminated. In this subsection, we show how to eliminate schizophrenic local declarations, which is more difficult. Clearly, as any statement, a local declaration can only be schizophrenic if it is left and entered at the same point of time. If this happens, we must distinguish between the two incarnations of the local variable, that may have different values. As both incarnations exist in the same macrostep, we need to store one of these values in a copy of the variable. Some sophisticated techniques – again based on microsteps rather than on macrosteps – are necessary to compute these different values.

For example, consider module *LocalReincarnation* given in Figure 3 (which is adapted from P17 in [2], page 132). The intended behavior of this module is as follows: We first enter the loop, and also the local variable declaration. At that instant of time, x can certainly not be present, since there is no emission for x. We therefore emit *xOff* in the first conditional statement and reach with the next microstep location ℓ, which completes this macrostep. In the next macrostep, we first emit x, and therefore emit *xOn* in the following conditional. In the next microstep, we leave the scope of the declaration of x, so that the value of this incarnation of x will be forgotten. At the same instant of time, the next microstep starts a new loop body, thus creating a new local declaration with a new incarnation of x. Of course, this new incarnation has nothing to do with the previous one. In particular, it is not present, since there is no emission for it. Hence, we also emit *xOff* in the first conditional and reach then location ℓ.

Hence, the module will emit *xOff* in the first macrostep, and afterwards both *xOn* and *xOff*, which seems to be contradictory since at any point of time, any variable must have one and only one value. However, this is not a contradiction, since there is not *one* variable x, but two, namely the old and the new incarnation of x.

A statement may even have more than one incarnation in a single macrostep, which is shown with module *MultipleReincarnation*$_2$ of Figure 4 (it is an adaptation of P18 of [2], page 138). The intended behavior is quite complicated and is described in detail in [2] (consider also Figure 6). The interesting issue is that after starting time, the body of the innermost loop (drawn in the box) yields three incarnations, with different values of the local variables. For this reason, three incarnations of the conditional statement in the box are

```
module MultipleReincarnation_2 :
output x_12^00, x_12^01, x_12^10, x_12^11;
do
  local x_1 in
    weak abort
        [ℓ_1 : pause; emit x_1]
    ||
      do
        local x_2 in
          weak abort
              [ℓ_2 : pause; emit x_2]
          ||
            do
              if x_1 then
                if x_2 then emit x_12^11
                else emit x_12^10 end
              else
                if x_2 then emit x_12^01
                else emit x_12^00 end
              end;
              ℓ_0 : pause
            while 1
          when x_2
        end local
      while 1
    when x_1
  end local
while 1
end module
```

Figure 4: Multiple Reincarnations.

concurrently executed. Each one emits one variable, so that x_{12}^{11}, x_{12}^{10}, and x_{12}^{00} are all present after starting time.

Module *MultipleReincarnation*$_2$ can be extended in an obvious way to *MultipleReincarnation*$_3$, and so on. By induction, we can prove that the innermost local variable of *MultipleReincarnation*$_n$ will have n incarnations, its next local variable will have $n-1$ incarnations, and so on.

Hence, a statement can be arbitrarily often started at the same point of time. Consequently, several incarnations of the surface may be simultaneously executed with its depth.

For this reason, it seems that we need more than one copy of a local variable to store the values of all of its incarnations. Indeed, the solution for handling schizophrenic local variables as suggested in [2], generates n copies of a local variable, where n is the so-called *incarnation level*: this is the number of *local declarations* and *parallel statements* in which its declaration is enclosed. For example, in *MultipleReincarnation*$_2$, we reach the incarnation level 4, and the declaration of x_2 is at level 2. According to the incarnation level n of a local variable declaration, the translation given in [2] creates n copies of that variable. The interaction of these copies is very complicated [2].

In the following, we will present a different solution to correctly translate statements with schizophrenic local variables, where *only one copy of the local variable is sufficient*. However, our solution still requires to copy surfaces multiple times according to their nestings in *loops* and *sequences*. Note that this is not too problematic, since these copies refer to combinatorial gates only, and therefore do not introduce further states to the control flow automaton. This is very important for hardware and software code generation,

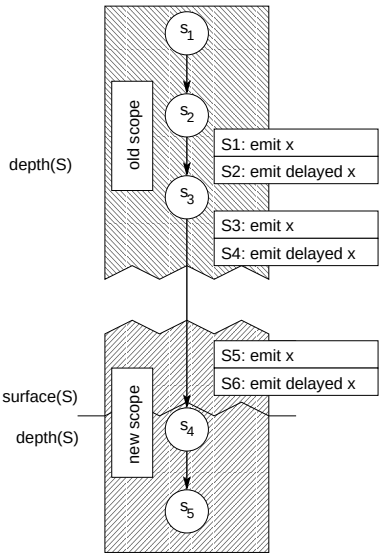

Figure 5: The reincarnation problem.

in particular when automaton code is generated to speed up worst case execution time.

In general, we have the situation pictured in Figure 5. Circles in Figure 5 represent states of the program, and the arrows between them represent macrosteps of the execution. The gray shaded area is moreover the scope of a local variable declaration x. Considering Figure 5, we can identify the following situations:

- Clearly, immediate emissions of a macrostep that is completely inside the scope like S_1 are uncritical.
- Delayed emissions of such a macrostep like S_2 should however only be seen in the old scope, but not in a possible new incarnation. Hence, the emission of S_2 must not be seen in the surface of the new scope, but in the depth of the old one.
- The same holds for immediate emissions of the depth of the old scope like S_3. They must not be seen in the surface of the new scope, but must be seen in the depth of the old one.
- Delayed emissions in the old scope that occur at termination time like S_4, have no effect at all, and can thus be ignored.
- Immediate emissions in the surface like S_5 of the new scope must be seen there, but not in the depth of the old scope.
- Delayed emissions in the surface like S_6 must not be seen there, also not in the depth of the old scope, but of course in the depth of the new scope.

The above problems indicate that it is reasonable to split the body S of a local declaration **local** x **in** S **end** into its surface and depth. If no further local declaration is included in S (this can be achieved by a bottom up traversal), and if all strong abortion statements have also been split into their surfaces and depths, we already know that this transformation neither changes the control nor the data flow. Hence, we consider now the equivalent statement

$$\textbf{local}\ x\ \textbf{in}\ \mathsf{surface}\,(S)\,;\mathsf{depth}\,(S)\ \textbf{end}$$

In the following, we use a new variable x_0 to hold the value that is generated in the surface, while the original variable x is used to hold the value generated in the depth. For this reason, we replace in $\mathsf{surface}\,(S)$ all occurrences of x with x_0, except for delayed emissions of x.

```
module MultipleReincarnation'_2 :
output x^00_12, x^01_12, x^10_12, x^11_12;
local x_1, x_1,0, x_2, x_2,0 in
  do
    if x_1,0 then
      if x_2,0 then emit x^11_12
      else emit x^10_12 end
    else
      if x_2,0 then emit x^01_12
      else emit x^00_12 end
    end;
    weak abort
        [ℓ_1 : pause; emit x_1]
      ||
        do
          weak abort
              [ℓ_2 : pause; emit x_2]
            ||
              do
                ℓ_0 : pause;
                if x_1 then
                  if x_2 then emit x^11_12
                  else emit x^10_12 end
                else
                  if x_2 then emit x^01_12
                  else emit x^00_12 end
                end
              while 1
          when x_2;
          if x_1 then
            if x_2,0 then emit x^11_12
            else emit x^10_12 end
          else
            if x_2,0 then emit x^01_12
            else emit x^00_12 end
          end
        while 1
    when x_1
  while 1
end module
```

Figure 6: Transformation of *MultipleReincarnation*$_2$

This already solves many of the above problems, however, the transformation is not yet complete. Nevertheless, all problems that appear in module *MultipleReincarnation*$_2$ are already solved with this simple transformation. The result, which is also the result of our complete solution outlined below, is given in Figure 6 (again, we have simplified the code). As can be seen, copies $x_{1,0}$ and $x_{2,0}$ of the local variables x_1 and x_2 have been generated, and the bodies of the local declarations have been transformed in sequences of their surfaces and depths.

As the depth of a loop generates a copy of the surface of its body, and the depth of a sequence $S_1; S_2$ generates a copy of the surface of S_1, some surfaces are copied in our approach, too. Different copies refer to different incarnations, which reveals the behavior of module *MultipleReincarnation*$_2$: Considering Figure 6, we see that our transformation has generated two copies of the conditional statement and each one is responsible for one emission: The copy with $x_{1,0}$ and $x_{2,0}$ emits x^{00}_{12}, the copy with x_1 and $x_{2,0}$ emits x^{10}_{12}, and the original statement with x_1 and x_2 emits x^{11}_{12}.

In general, the duplication of the surfaces can not be avoided: Without them, it is not possible to generate correct code for modules with schizophrenia problems.

The transformation we have outlined so far is not yet complete. Consider again a statement **local** x **in** S **end**. Assume S is transformed into $\mathsf{surface}\,(S)$; $\mathsf{depth}\,(S)$, and x is replaced by x_0 in $\mathsf{surface}\,(S)$ (except for delayed emissions). If the statement is started, its surface is instantaneously executed, and $\mathsf{depth}\,(S)$ is entered. Hence, we may have to evaluate some conditions of conditional statements in $\mathsf{depth}\,(S)$. If such a condition refers to x, we should refer to x_0 instead, which is however not the case with our so far incomplete transformation. As the same condition may be also evaluated not at starting time, we can not simply replace x with x_0.

Hence, to complete our translation, we have to *adapt some* of the conditions in the depth part. This completes our transformation: Note that no assignment of the depth will use x_0, since the depth simply performs no assignments at starting time. Hence, all assignments and emissions of the depth are already correct, and we can concentrate on evaluations of conditions for the control flow. The only basic statement that evaluates such a condition at starting time is the conditional statement. Hence, it remains to change x into x_0 in some of the conditions of conditional statements of the depth. Note that only those conditions must be replaced that are evaluated at starting time (which depends on the current inputs). For this reason, we apply the following transformation to the depth, which moreover disables delayed emissions at termination time:

DEFINITION 4 (FIXING SURFACE CONDITIONS).
Given a Quartz *statement S without local signal declarations, we define the statement* $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, S)$ *of S for a termination condition t and a precondition φ as follows:*

- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{nothing}) :\equiv \textbf{nothing}$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{emit}\ x) :\equiv \textbf{emit}\ x$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{emit delayed}\ x)$
  $:\equiv \begin{cases} \textbf{if}\ \neg t\ \textbf{then emit delayed}\ s\ \textbf{end} & \textit{:if } x \equiv s \\ \textbf{emit delayed}\ x & \textit{:if } x \not\equiv s \end{cases}$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, y := \tau) :\equiv y := \tau$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, y := \textbf{delayed}\ \tau)$
  $:\equiv \begin{cases} \textbf{if}\ \neg t\ \textbf{then}\ s := \textbf{delayed}\ \tau\ \textbf{end} & \textit{:if } y \equiv s \\ y := \textbf{delayed}\ \tau & \textit{:if } y \not\equiv s \end{cases}$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \ell : \textbf{pause}) :\equiv \ell : \textbf{pause}$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{if}\ \sigma\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})$
  $:\equiv$ **if** $\varphi \wedge [\sigma]^{s_0}_s \vee \neg\varphi \wedge \sigma$
  then $\mathsf{surfcond}^{s_0}_s\left(t, \varphi \wedge [\sigma]^{s_0}_s, S_1\right)$
  else $\mathsf{surfcond}^{s_0}_s\left(t, \varphi \wedge \neg\,[\sigma]^{s_0}_s, S_2\right)$
  end
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, S_1; S_2)$
  $:\equiv\ \mathsf{surfcond}^{s_0}_s\,(t, \varphi, S_1)\,;$
  $\mathsf{surfcond}^{s_0}_s\left(t, \varphi \wedge [\mathsf{inst}\,(S_1)]^{s_0}_s, S_2\right)$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, S_1 \parallel S_2)$
  $:\equiv \mathsf{surfcond}^{s_0}_s\,(t, \varphi, S_1) \parallel \mathsf{surfcond}^{s_0}_s\,(t, \varphi, S_2)$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{do}\ S\ \textbf{while}\ \sigma)$
  $:\equiv \textbf{do}\ \mathsf{surfcond}^{s_0}_s\,(t, \varphi, S)\ \textbf{while}\ \sigma$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{suspend}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \textbf{suspend}\ \mathsf{surfcond}^{s_0}_s\,(t, \varphi, S)\ \textbf{when}\ \sigma$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{weak suspend}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \textbf{weak suspend}\ \mathsf{surfcond}^{s_0}_s\,(t, \varphi, S)\ \textbf{when}\ \sigma$
- $\mathsf{surfcond}^{s_0}_s\,(t, \varphi, \textbf{abort}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \textbf{abort}\ \mathsf{surfcond}^{s_0}_s\,(t, \varphi, S)\ \textbf{when}\ \sigma$

- $\mathsf{surfcond}_s^{s_0}\,(t, \varphi, \textbf{weak abort}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \textbf{weak abort}\ \mathsf{surfcond}_s^{s_0}\,(t, \varphi, S)\ \textbf{when}\ \sigma$
- $\mathsf{surfcond}_s^{s_0}\,(t, \varphi, \textbf{local}\ x\ \textbf{in}\ S\ \textbf{end})$
  $:\equiv \textbf{local}\ x\ \textbf{in}\ \mathsf{surfcond}_s^{s_0}\,(t, \varphi, S)\ \textbf{end}$
- $\mathsf{surfcond}_s^{s_0}\,(t, \varphi, \textbf{local}\ y : \alpha\ \textbf{in}\ S\ \textbf{end})$
  $:\equiv \textbf{local}\ y : \alpha\ \textbf{in}\ \mathsf{surfcond}_s^{s_0}\,(t, \varphi, S)\ \textbf{end}$
- $\mathsf{surfcond}_s^{s_0}\,(t, \varphi, \textbf{now}\ \sigma) :\equiv \textbf{now}\ \sigma$
- $\mathsf{surfcond}_s^{s_0}\,(t, \varphi, \textbf{during}\ S\ \textbf{holds}\ \sigma)$
  $:\equiv \textbf{during}\ \mathsf{surfcond}_s^{s_0}\,(t, \varphi, S)\ \textbf{holds}\ \sigma$

Intuitively, in the above definition for $\mathsf{surfcond}_s^{s_0}\,(t, \varphi, S)$, it is assumed that the precondition φ describes all conditions where the execution up to S will not take time (some further execution of S may also not take time). Hence, if φ holds, we have to replace s with its surface value s_0 to obtain $[\sigma]_s^{s_0}$, otherwise the condition σ is left unchanged. Note that the expression $\varphi \wedge [\sigma]_s^{s_0} \vee \neg\varphi \wedge \sigma$ either corresponds to the substituted condition $[\sigma]_s^{s_0}$ or to the original one σ, depending on whether the expression is evaluated at starting time (which is encoded by φ). *To summarize, a local declaration* **local** x **in** S **end** *that can be reached with the precondition φ is transformed into the following sequence:*

$$\begin{array}{l}\mathsf{subst\_immed}(\mathsf{surface}\,(S)\,, x, x_0);\\ \mathsf{surfcond}_x^{x_0}\,(\mathsf{term}\,(S)\,, \varphi, \mathsf{depth}\,(S))\end{array}$$

where subst_immed *is meant to replace all occurrences of x in S by x_0 except for delayed emissions of x or assignments to x.* We can now convince ourselves that this transformation behaves like S, but does no longer suffer from schizophrenia problems:

- Immediate emissions in the surface (like S_5 in Figure 5) are transformed into emissions of the new variable x_0, and are therefore only seen in the surface and in conditions of the depth that are evaluated at starting time.
- Delayed emissions in the surface (like S_6 in Figure 5) are not changed, thus they are correctly seen in the depth.
- Immediate emissions in the depth that are not executed at termination time of the depth (like S_1 in Figure 5) are not changed. If the macrostep is entirely contained in a local declaration, then it can never overlap with surfaces of other incarnations. Also delayed emissions (like S_2 in Figure 5) are not changed, even if the statement may terminate at the next point of time. If it would terminate, then the surface of a new incarnation will refer to x_0 and not to x.
- Immediate emissions of the depth at termination time of the depth (like S_3 in Figure 5) are seen there, but neither in the possibly overlapping surface of a new incarnation, nor in the conditions of the depth in the new scope that are evaluated at this point of time, since these parts refer to x_0.
- Delayed emissions of x or assignments to x in the depth at termination time (like S_4 in Figure 5) have been disabled by embracing them in a conditional **if** $\neg t$ **then** ... **end**, where t hold exactly at termination time.

Hence, all situations are correctly handled, so that the obtained statement does no longer suffer from schizophrenia problems. A similar transformation is applied to local declarations of state variables. Hence, we now end up with the following definition:

DEFINITION 5 (ELIMINATING SCHIZOPHRENIA). *For any* Quartz *statement S, we define a corresponding statement that is obtained by rewriting with the following equations during a bottom-up traversal over the syntax tree of S, where* $\mathsf{subst\_immed}(P, x, x_0)$ *is supposed to replace x by x_0 in P except in delayed emissions of x or delayed assignments to x:*

- $\textbf{abort}\ S\ \textbf{when}\ \sigma :\equiv \mathsf{surface}\,(S)\,;\ \textbf{abort}\ \mathsf{depth}\,(S)\ \ \textbf{when}\ \sigma$
- $\textbf{local}\ x\ \textbf{in}\ S\ \textbf{end} :\equiv \begin{bmatrix}\mathsf{subst\_immed}(\mathsf{surface}\,(S)\,, x, x_0);\\ \mathsf{surfcond}_x^{x_0}\,(\mathsf{term}\,(S)\,, \varphi, \mathsf{depth}\,(S))\end{bmatrix}$

The equation of the local declaration is used both for local event and state variables. Note that the above transformation eliminates local declarations (recall that we have not allowed shadowing of local variables, i.e., all local variable names must be unique). The main theorem of this paper is then the following one:

THEOREM 3 (ELIMINATING SCHIZOPHRENIA PROBLEMS). *The transformation described in definition 3 yields an equivalent statement that has no schizophrenia problems.*

4. ALGORITHMS FOR COMPILATION

In the previous section, we have developed program transformations to transform synchronous programs into equivalent ones without schizophrenia problems. In this section, we list a complete algorithm for translating synchronous programs into equivalent equation systems and analyse its complexity. Of course, the algorithm applies our transformation to eliminate schizophrenia problems during the translation. The algorithm given in Figure 7 is based on an improvement of the circuit synthesis that has been presented in [15]. We have proved the equivalence of the older circuit synthesis presented in [15], the one given in Figure 7, and the semantics given via the control flow predicates in [16] with the interactive theorem prover HOL [9]. In particular, we have proved the following theorem (the second assumption is an invariant of the construction):

THEOREM 4 (TRANSLATION TO EQUATION SYSTEMS). *Given any* Quartz *statement S, and Boolean expressions st, φ, sp, and kl such that the following conditions hold at any time:*

- *for any loop '***do** P **while** x*' in S, we have* $\neg\mathsf{inst}\,(P)$
- $st \wedge \mathsf{in}\,(S) \rightarrow \neg sp \wedge (\mathsf{term}\,(S) \vee kl)$

Abbreviate $S_0 :\equiv \textbf{suspend abort}\ S\ \textbf{when}\ kl\ \textbf{when}\ sp$*, and let S' be the statement that is obtained from S_0 by the elimination of schizophrenia problems (definition 5) and the elimination of nondeterministic statements by introducing control variables [16]. Then, the functions given in Figure 7 compute a tuple* $(C, A, R, I, T, G_s, G_d, S_s, S_d) = \mathsf{EQS}\,(\{\}, st, \varphi, sp, kl, S)$ *with the following properties:*

- *C is the set of control variables used to compute S'*
- $A = \mathsf{in}\,(S')$
- $I = \mathsf{inst}\,(S')$
- $T = \mathsf{term}\,(S')$
- $G_s = \mathsf{gcmd}\,(\varphi, \mathsf{surface}\,(S'))$
- $G_d = \mathsf{gcmd}\,(\varphi, \mathsf{depth}\,(S'))$
- $S_s = \mathsf{surface}\,(S')$ *and* $S_d = \mathsf{depth}\,(S')$
- $\mathcal{R}_{\mathsf{cf}}(st, S') = \bigwedge_{\tau \in R} \tau$

The above assumptions mean that (1) for all inputs, all loop bodies must not be instantaneous, and (2) if the start signal st occurs when the control flow is already inside the statement, then the control flow currently leaves the statement, i.e., there is no suspension and the statement either terminates on its own or is killed by the kl signal. Hence, already active statements must not be started, unless they currently terminate.

As '**suspend abort** S **when** 0 **when** 0' is equivalent to S, it follows that the above theorem allows us to compute the transition

function EQS $(C, st, \varphi, sp, kl, P)$
  $N :=$ **nothing**;
  case P **of**
    nothing : **return** $(C, 0, \{\}, 1, 0, \{\}, \{\}, P, P)$;
    emit x, **emit delayed** x, $y := \tau$, $y :=$ **delayed** τ, **now** σ :
      return $(C, 0, \{\}, 1, 0, \{(\varphi, P)\}, \{\}, P, N)$;
    ℓ : **pause** :
      return $(C, \ell, \{\mathsf{X}\ell = st \vee sp \wedge \ell\}, 0, \ell, \{\}, \{\}, N, P)$;
    if σ **then** S_1 **else** S_2 **end** :
      return cond_EQS $(C, st, \varphi, sp, kl, \sigma, S_1, S_2)$;
    choose $S_1 \,[]\, S_2$ **end** :
      $c :=$ **newvar**$()$;
      return cond_EQS $(C \cup \{c\}, st, \varphi, sp, kl, c, S_1, S_2)$;
    $S_1; S_2$: **return** seq_EQS $(C, st, \varphi, sp, kl, S_1, S_2)$;
    $S_1 \parallel S_2$: **return** par_EQS $(C, st, \varphi, sp, kl, S_1, S_2)$;
    $S_1 \,|||\, S_2$:
      $c_1 :=$ **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**$(P_1, P_2), \sigma)$;
      return EQS $(C \cup \{c_1, c_2\}, st, \varphi, sp, kl, P)$;
    do S **while** σ :
      return dowhile_EQS $(C, st, \varphi, sp, kl, \sigma, S)$;
    suspend S **when** σ :
      return susp_EQS $(C, st, \varphi, sp, kl, \sigma, S, 0)$;
    weak suspend S **when** σ :
      return susp_EQS $(C, st, \varphi, sp, kl, \sigma, S, 1)$;
    abort S **when** σ :
      return abort_EQS $(C, st, \varphi, sp, kl, \sigma, S, 0)$;
    weak abort S **when** σ :
      return abort_EQS $(C, st, \varphi, sp, kl, \sigma, S, 1)$;
    local x **in** S **end**, **local** $x : \alpha$ **in** S **end** :
      return local_EQS $(C, st, \varphi, sp, kl, x, S)$;
    during S **holds** σ :
      $(C, A, R, I, T, G_s, G_d, S_s, S_d) :=$ EQS $(C, st, \varphi, sp, kl, S)$;
      return $(C, A, R, I, T, G_s, G_d \cup \{(A,$ **now** $\sigma)\}, S_s, S_d)$;
  end case ;
end function

function local_EQS $(C_0, st, \varphi, sp, kl, x, S)$
  $(C, A, R, I, T, G_s, G_d, S_s, S_d) :=$ EQS $(C_0, st, \varphi, sp, kl, S)$;
  $x_0 :=$ **newvar**$()$;
  $S_s :=$ **subst_immediate**(S_s, x, x_0);
  $S_d :=$ surfcond$_x^{x_0}\,(T, \varphi, S_d)$;
  return seq_EQS $(C_0, st, \varphi, sp, kl, S_s, S_d)$;
end function

function par_EQS $(C_0, st, \varphi, sp, kl, S_1, S_2)$
  $(C_1, A_1, R_1, I_1, T_1, G_s^1, G_d^1, S_s^1, S_d^1)$
    $:=$ EQS $(C_0, st, \varphi, sp, kl, S_1)$;
  $(C_2, A_2, R_2, I_2, T_2, G_s^2, G_d^2, S_s^2, S_d^2)$
    $:=$ EQS $(C_1, st, \varphi, sp, kl, S_2)$;
  $A := A_1 \vee A_2$; $R := R_1 \cup R_2$; $I := I_1 \wedge I_2$;
  $T := T_1 \wedge \neg A_2 \vee T_2 \wedge \neg A_1 \vee T_1 \wedge T_2$;
  $S_s :=$ **mk_par**(S_s^1, S_s^2);
  $S_d :=$ **mk_par**(S_d^1, S_d^2);
  return $(C_2, A, R, I, T, G_s^1 \cup G_s^2, G_d^1 \cup G_d^2, S_s, S_d)$;
end function

function cond_EQS $(C_0, st, \varphi, sp, kl, \sigma, S_1, S_2)$
  $st_1 := st \wedge \sigma$; $\varphi_1 := \varphi \wedge \sigma$;
  $st_2 := st \wedge \neg\sigma$; $\varphi_2 := \varphi \wedge \neg\sigma$;
  $(C_1, A_1, R_1, I_1, T_1, G_s^1, G_d^1, S_s^1, S_d^1)$
    $:=$ EQS $(C_0, st_1, \varphi_1, sp, kl, S_1)$;
  $(C_2, A_2, R_2, I_2, T_2, G_s^2, G_d^2, S_s^2, S_d^2)$
    $:=$ EQS $(C_1, st_2, \varphi_2, sp, kl, S_2)$;
  $A := A_1 \vee A_2$; $R := R_1 \cup R_2$; $T := T_1 \vee T_2$;
  $I := \sigma \wedge I_1 \vee \neg\sigma \wedge I_2$;
  $S_s :=$ **mk_cond**(σ, S_s^1, S_s^2);
  $S_d :=$ **mk_cond**(σ, S_d^1, S_d^2);
  return $(C_2, A, R, I, T, G_s^1 \cup G_s^2, G_d^1 \cup G_d^2, S_s, S_d)$;
end function

function seq_EQS $(C_0, st, \varphi, sp, kl, S_1, S_2)$
  $(C_1, A_1, R_1, I_1, T_1, G_s^1, G_d^1, S_s^1, S_d^1)$
    $:=$ EQS $(C_0, st, \varphi, sp, kl, S_1)$;
  $st_2 := st \wedge I_1 \vee \neg sp \wedge \neg kl \wedge T_1$;
  $\varphi_2 := I_1 \wedge \varphi \vee T_1$;
  $(C_2, A_2, R_2, I_2, T_2, G_s^2, G_d^2, S_s^2, S_d^2)$
    $:=$ EQS $(C_1, st_2, \varphi_2, sp, kl, S_2)$;
  $T := T_1 \wedge I_2 \vee T_2$;
  $G_s := G_s^1 \cup \{(\gamma \wedge I_1, \alpha) \mid (\gamma, \alpha) \in G_s^2\}$;
  $G_d := G_d^1 \cup G_d^2 \cup \{(\gamma \wedge A_1, \alpha) \mid (\gamma, \alpha) \in G_s^2\}$;
  $S_s :=$ **mk_seq**$(S_s^1,$ **mk_cond**$(I_1, S_s^2,$ **nothing**$))$;
  $S_d :=$ **mk_seq**$(S_d^1,$ **mk_cond**$(A_1, S_s^2,$ **nothing**$), S_d^2)$;
  return $(C_2, A_1 \vee A_2, R_1 \cup R_2, I_1 \wedge I_2, T, G_s, G_d, S_s, S_d)$;
end function

function dowhile_EQS $(C_0, st, \varphi, sp, kl, \sigma, S)$
  $t :=$ **newvar**$()$;
  $st' := st \vee \neg sp \wedge \neg kl \wedge \sigma \wedge t$; $\varphi' := \varphi \vee t \wedge \sigma$;
  $(C, A, R, I', T', G_s, G_d, S_s, S_d) :=$ EQS $(C_0, st', \varphi', sp, kl, S)$;
  $R := \{[\tau]_t^{T'} \mid \tau \in R\}$; $I := [I']_t^{T'}$; $T := T' \wedge \neg\sigma$;
  $G_s := \{([\gamma]_t^{T'}, \alpha) \mid (\gamma, \alpha) \in G_s\}$;
  $G_d := \{([\gamma]_t^{T'}, \alpha) \mid (\gamma, \alpha) \in G_d\}$
    $\cup \{([\gamma]_t^{T'} \wedge T' \wedge \sigma, \alpha) \mid (\gamma, \alpha) \in G_s\}$;
  $D :=$ **mk_seq**$(S_d,$ **mk_cond**$(\sigma, S_s,$ **nothing**$))$;
  $S_d :=$ **mk_dowhile**(D, σ);
  return $(C, A, R, I, T, G_s, G_d, S_s, S_d)$;
end function

function susp_EQS $(C_0, st, \varphi, sp, kl, \sigma, S, wk)$
  $sp' := sp \vee \sigma \wedge \neg kl$;
  $(C, A, R, I, T, G_s, G_d, S_s, S_d) :=$ EQS $(C_0, st, \varphi, sp', kl, S)$;
  if $\neg wk$ **then** $G_d := \{(\gamma \wedge \neg\sigma, \alpha) \mid (\gamma, \alpha) \in G_d\}$ **end**;
  $S_d :=$ **mk_suspend**(wk, S_d, σ);
  return $(C, A, R, I, T \wedge \neg\sigma, G_s, G_d, S_s, S_d)$;
end function

function abort_EQS $(C_0, st, \varphi, sp, kl, \sigma, S, wk)$
  $kl' := kl \vee \sigma$;
  $(C, A, R, I, T, G_s, G_d, S_s, S_d) :=$ EQS $(C_0, st, \varphi, sp, kl', S)$;
  if $\neg wk$ **then** $G_d := \{(\gamma \wedge \neg\sigma, \alpha) \mid (\gamma, \alpha) \in G\}$ **end**;
  $S_d :=$ **mk_abort**(wk, S_d, σ);
  return $(C, A, R, I, T \vee A \wedge \sigma, G_s, G_d, S_s, S_d)$ **end**;
end function

Figure 7: Translation of Quartz **Statements to Equivalent Equation Systems**

relation $\mathcal{R}_{\mathsf{cf}}(st, S)$ of S in form of an equation system. Note further that the condition that a start signal should only occur when the control currently leaves the statement is simply achieved by using the modified start signal $st' := st \wedge (\mathsf{in}(S) \rightarrow \mathsf{term}(S))$, where st is the original start signal.

Finally, let us consider the complexity of the translation. For this reason, we define $L_s(n)$ and $L_d(n)$ as the maxima of the lengths of statements $\mathsf{surface}(S)$ and $\mathsf{depth}(S)$, respectively, where S ranges over the Quartz statements S of length $\leq n$. In [16], it is proved that all control flow predicates can be computed in time $O(|S|)$ with some sharing of common subterms that is achieved by abbreviating the predicates by new variables.

Considering the definition of $\mathsf{surface}(S)$, it is then easily seen that $L_s(n+1) \leq L_s(n) + C_1$ and $L_d(n+1) \leq L_d(n) + L_s(n) + n + C_3$ and $L_d(n+1) \leq L_d(n) + L_s(n) + C_2$ holds. As the only function obeying the recurrence relation $f(n+1) = \alpha f(n) + g(n)$ is $f(n) = \alpha^n f(0) + \sum_{i=0}^{n-1} \alpha^{n-1-i} g(i)$, it follows that $L_s(n)$ is of order $O(n)$, and that $L_d(n)$ is of order $O(n^2)$. Hence, we have the following result:

LEMMA 1 (SIZE OF SURFACE AND DEPTH). *Given any basic* Quartz *statement S, it follows for the statements* $\mathsf{surface}(S)$ *and* $\mathsf{depth}(S)$ *that their sizes are of orders $O(|S|)$ and $O(|S|^2)$, respectively, provided that the control flow predicates are abbreviated as variables.*

To analyse the entire complexity, note first that EQS runs in time $O(|S|^2)$ in case that S contains no local declaration (proof by induction). If surfaces were not copied, and are shared by using pointers to them instead, this can even be done in time $O(|S|)$. A proof for the general case is however very subtle and depends on the chosen data structures. However, we claim that there is an implementation that runs in time $O(|S|^2)$, which is also the complexity that is stated in [2].

PROPOSITION 1. *For any basic* Quartz *statement S, there is a translation procedure that runs in time $O(|S|^2)$.*

5. EXPERIMENTAL RESULTS

We have embedded the language Quartz in the theorem prover HOL [9], and have proved the correctness of the transformations of this paper. Moreover, we have implemented a compiler for Quartz in Java that is based on the transformations of this paper and the circuit synthesis given in [15]. For a given Quartz program, our compiler either outputs an equation system or a file containing sc code [8], which is one of the intermediate formats of the Esterel compiler. Hence, we can use the code generators and optimizers of the Esterel design tools. In particular, we can also use the very sophisticated Esterel tools for causality analysis.

For a generic version of module *MultipleReincarnation*$_n$ (cf. Figure 4), we obtained the results that are listed in table 1. Note that the length of *MultipleReincarnation*$_n$ is of order $O(2^n)$, so that the obtained C programs are essentially of the same size. The numbers in the table are thereby the numbers of Boolean gates (and, or, and not gates) that are generated in the circuit synthesis. As can be seen, the size of the sc code generated from our compiler is only about 30% compared to the one generated with the Esterel compiler, and it still remains smaller after optimization during translation to ssc code with the Esterel compiler.

	Quartz		Esterel	
n	**sc**	**ssc**	**sc**	**ssc**
1	48	29	71	39
2	81	51	149	83
3	133	86	268	144
4	218	144	466	233
5	368	247	827	372
6	647	439	1535	605
7	1183	808	2920	1020
8	2232	1530	6056	1793
9	4306	2957	12589	3276
10	8429	5793	26557	6173

Table 1: Comparison of the code size (number of gates) obtained by synthesizing a generic version of module *MultipleReincarnation*$_n$ (Figure 4)

6. REFERENCES

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

[2] G. Berry. The constructive semantics of pure Esterel, July 1999.

[3] G. Berry. The Esterel v5_91 language primer. http://www.esterel.org, June 2000.

[4] F. Boussinot. SugarCubes implementation of causality. Research Report 3487, INRIA, Sophia Antipolis Cedex, France, September 1998.

[5] J. Brzozowski and C.-J. Seger. *Asynchronous Circuits*. Springer Verlag, 1995.

[6] Cadence Design Systems,Inc. Website, 2000. http://www.cadence.com.

[7] Esterel Web. Website, 2000. http://www.esterel.org.

[8] A. Girault and G. Berry. Circuit generation and verification of Esterel programs. Research report 3582, INRIA, December 1998.

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

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

[11] Jester Home Page. Website, 2000. http://www.parades.rm.cnr.it/projects/jester/jester.html.

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

[13] S. Malik. Analysis of cycle combinational circuits. *IEEE Transactions on Computer Aided Design*, 13(7):950–956, July 1994.

[14] POLIS Homepage, 2000. http://www-cad.eecs.berkeley.edu/

[15] K. Schneider. A verified hardware synthesis for Esterel. In F. Rammig, editor, *International IFIP Workshop on Distributed and Parallel Embedded Systems*, pages 205–214. Kluwer Academic Publishers, 2000.

[16] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *International Conference on Application of Concurrency to System Design (ICACSD 2001)*. IEEE Computer Society Press, June 2001. http://goethe.ira.uka.de/fmg/ps/Schn01a.ps.gz.

[17] T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference (EDTC)*, Paris, France, 1996. IEEE Computer Society Press.