A VERIFIED HARDWARE SYNTHESIS OF ESTEREL PROGRAMS

Klaus Schneider

Institute for Computer Design and Fault Tolerance (Prof. Dr.-Ing. D. Schmid)
University of Karlsruhe, P.O. Box 6980, 76128 Karlsruhe
Klaus.Schneider@informatik.uni-karlsruhe.de
http://goethe.ira.uka.de/~schneider

Synchronous programming languages like Esterel *are becoming more and more popular for the design of multi-threaded reactive systems. We have embedded a variant of the* Esterel *language in the interactive theorem prover* HOL *so that we can formally reason about programs of the language and – at a meta level – about the language itself. Based on a separation of the control and data flow of the programs, we have defined a new translation to equation systems. Our new translation is simpler than state-of-the-art translations, and it does not suffer from the schizophrenia problems of parallel statements. Furthermore, we have proved the correctness of our translation with* HOL, *so that* HOL *can be used for formal synthesis.*

1 INTRODUCTION

Synchronous languages are becoming more and more attractive [1, 2, 3, 4, 5] for the design and the formal verification of parallel reactive real-time systems. There are imperative languages like Esterel [6, 7, 1], data flow languages [9], and graphical languages like some Statechart variants [10, 11]. We concentrate in this paper on imperative languages, but emphasize that graphical and imperative languages can be naturally translated into each other [11].

The basic paradigm of (imperative) synchronous languages is the *perfect synchrony*, which means that most of the statements are executed in zero time (at least in the ide-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35409-5_23

B. Kleinjohann (ed.), *Architecture and Design of Distributed Embedded Systems*
© IFIP International Federation for Information Processing 2001

alized programmer's model). Consumption of time must be explicitly programmed with special statements, as e.g. the **pause** statement in Esterel. Each execution of a **pause** statement consumes one logical unit of time, and therefore separates different interactions. As the **pause** statement is the only basic statement that consumes time, it follows that all threads of a synchronous program run in lockstep: they execute the code between **pause** statements in zero time, and synchronize at the next **pause** statements. Note that this synchronization is simply due to the semantics of the language.

The control flow of a synchronous program $\mathcal{P}$ can therefore be compiled into a finite state machine $\mathcal{A}_\mathcal{P}$ in that we describe how the control flow moves from a set of currently active **pause** statements to the set of **pause** statements that are active at the next point of time. Of course, we must also consider the data flow of a program, i.e. how the transition of the control flow manipulates the data values of the program. Therefore, we can model any Esterel program by a finite state control flow that manipulates possibly infinite data types. The control and data flow can be described in form of equation systems that can be furthermore converted into a sequential (i.e. single-threaded) imperative program, as e.g. a C or Java program [1, 4, 5] or to a VHDL program to synthesize a hardware circuit. Therefore, Esterel programs can be both used for hardware or software generation.

The translation of synchronous programs to the corresponding equation system is an essential means for code generation and formal verification. Therefore, a lot of ways have been studied for this translation: [12] distinguishes between a process-algebraic, a finite-state machine, and a hardware circuit semantics. The process-algebraic and the finite state machine semantics are used to enumerate the control states of a program by a depth first traversal so that the control flow state machine is explicitly constructed. Therefore, these translations suffer from the drawback that a program of length n may have $O(n)$ **pause** statements and therefore $2^{O(n)}$ states.

Newer versions of Esterel compilers work more efficiently [12]: They translate a program of length n in polynomial time to an equivalent equation system that is expressed in terms of hardware circuits. Hence, one often speaks of a 'hardware synthesis', although this representation is used for software synthesis as well. Despite the fact that these equations might still define $O(2^n)$ states, the translation can be performed in polynomial time, since the equation system make use of a symbolic representation [13].

However, the circuit semantics as given as the 'basic translation' in [12] suffers from so-called schizophrenia problems that arise when a statement is terminated and restarted at the same point of time. This requires the reincarnation of local signals that then appear with different values at the same point of time. Moreover, circuit parts used to implement parallel statements (the synchronizer circuits) are erroneously used for the old and new incarnations of the loop body. Therefore, [12] suggests a more complicated hardware synthesis to overcome these problems.

Instead, we have defined a new 'basic translation' to equation systems to circumvent the schizophrenia problems of the control flow, i.e., for parallel statements. However, we must still handle schizophrenia problems for the data flow, i.e., for local signals. We present our translation in form of a hardware synthesis in section 3 for

our language Quartz. Quartz extends Esterel by several statements used to explicitly program nondeterminism and asynchronous parallel execution to *model* reactive systems. As these additional statements can not be simply translated to deterministic synchronous hardware circuits, we do not consider them in this paper. However, we consider the additional delayed data manipulating statements of Quartz.

To assure the correctness of our translation, we have embedded Quartz in the interactive theorem prover HOL [14], so that Quartz programs have become part of HOL's higher order logic. We have then defined the hardware synthesis, and have proved its correctness, which required to prove a couple of lemmata in advance. Based on the correctness theorem, we can now use the HOL theorem prover to implement a formal synthesis tool: the translation of a program can be done by HOL, where a correctness proof is additionally generated for the particular program. Furthermore, this can be performed very efficiently: the formal synthesis can be done in polynomial time, and our experimental results showed that it can even compete with standard compilers.

2 SYNTAX AND INFORMAL SEMANTICS

Synchronous languages like Esterel or Quartz are mainly concerned with the implementation of the complex control flow of threads while data types and expressions may be borrowed from a host language. Hence, we do neither consider types nor expressions in the following. Our embedding of Quartz in HOL does also directly borrow types and expressions from the HOL logic.

As time is modeled by the natural numbers $\mathbb{N}$, the semantics of an expression is a function of type $\mathbb{N} \to \alpha$ for some type α. In Quartz, we distinguish between *event* and *valued signals*. The semantics of an event signal is a function of type $\mathbb{N} \to \mathbb{B}$, while the semantics of a valued signal may have the more general type $\mathbb{N} \to \alpha$. Valued signals are 'sticky': they store their value until a data operation is applied. Event signals, on the other hand, are not sticky: if they are not explicitly made present at the next point of time, they will be reset to 0 (we denote boolean values as 1 and 0).

The basic statements of Quartz are given below:

Definition 1 (Basic Statements) *The following rules define the set of basic statements of* Quartz, *where it is assumed that* S, S_1, *and* S_2 *are also* Quartz *statements,* ℓ *is a label variable,* x *an event signal,* σ *a boolean expression, and* y *a valued signal:*

- **nothing** *(empty statement)*
- ℓ : **pause** *(consumption of one logical unit of time)*
- **emit** x *and* **emit delayed** x *(signal emissions)*
- $y := \tau$ *and* $y :=$ **delayed** τ *(assignments)*
- **if** σ **then** S_1 **else** S_2 **end** *(conditional)*
- S_1**;** S_2 *(sequential composition)*
- $S_1 \parallel S_2$ *(synchronous parallel composition)*
- **while** σ **do** S **end** *(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)*

Before giving a precise formal semantics in terms of our new hardware synthesis, we informally discuss the meaning of the above statements (for further explanations and examples, we refer to [7]). In general, a statement S is 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*.

pause is the only basic statement that consumes time. The statement does not affect any data values. Each **pause** statement of a program is endowed with a unique location variable ℓ that we will later use to encode the control flow of the programs. **nothing** is an empty statement: it simply does nothing, i.e. it does neither consume time, nor does it affect any data values. **emit** x immediately makes the event signal x present, i.e., the value of x at that point of time is then 1. Executing $y{:=}\tau$ will immediately change the value of y to the current value of the expression τ. The statements **emit delayed** x and $y :=$ **delayed** τ are similarly defined as **emit** x and $y := \tau$, respectively, but with a delay of one unit of time. 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 data manipulating statements consumes time, but they may affect the signal values at the next point of time.

if σ **then** S_1 **else** S_2 **end** is the conditional statement that checks whether the expression σ currently evaluates to 1 or 0 and then immediately either executes S_1 or S_2 (depending on the value of σ). $S_1; S_2$ 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 the execution of S_2.

$S_1 \parallel S_2$ denotes the synchronous parallel execution of S_1 and S_2: if $S_1 \parallel S_2$ is entered, we both enter S_1 and S_2 and immediately proceed with both executions. As long as both S_1 and S_2 are active, both threads are concurrently 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 so does $S_1 \parallel S_2$.

while σ **do** S **end** implements iteration: if this statement is entered, two cases are to be distinguished: If σ does not hold, then the statement instantaneously terminates. Otherwise, we immediately execute S. It is possible that S never terminates. However, if S terminates, and at that point of time σ holds again, then S is immediately restarted.

(weak) suspend S **when** σ implements process suspension, i.e. S is entered when the execution of this statement starts, regardless of the current value of σ. For the following points of time, the execution of S only proceeds if σ evaluates to 0, otherwise its execution is suspended until σ allows a further execution.

(weak) abort S **when** σ implements process abortion: S is immediately entered, regardless of the current value of σ. S is then executed as long as σ is 0. If σ becomes 1 during the execution of S, then S is immediately aborted. Hence, **abort** S **when** σ can either 'normally' terminate (when the execution of S terminates), or it may terminate by abortion (when σ enforces this).

The 'weak' variants of process suspension and abortion differ only on the treatment of the data manipulations at suspension or abortion time: while the strong variants ignore all data manipulation, all of them take place in the weak variants. There are also **immediate** variants of suspension and abortion that do also consider the value of the condition σ at starting time. These can be defined in terms of the other statements. There are also a lot of other statements that can be defined as macro expansions of basic statements.

Esterel has the same basic statements as the ones above. Esterel does also have event signals, as defined here. Valued signals of Quartz correspond to Esterel's valued signals, but we omit the status of these signals, as the status can be implemented by additional event signals, if wanted. Finally, Esterel has variables, which can take more than one value at a point of time. We do not consider these variables in this paper, although they do appear in Quartz as well.

3 TRANSLATING Qu rtz PROGRAMS TO EQUATION SYSTEMS

In this section, we present our new translation of Quartz and Esterel programs to equivalent equation systems. Similar to the state-of-the-art translation [12], our new translation is based on a recursive translation, where each program statement is implemented by a hardware circuit template. In contrast to [12], we do however distinguish between the control and the data flow of the program. Moreover, the hardware circuits we use have different inputs and outputs, that allow a more efficient translation. In particular, we do not need synchronizer circuits for the translation of parallel statements, and hence, our translation does not suffer from the subsequent schizophrenia problems. In the following three subsections, we first define the control flow, then the data flow, and finally combine both to a single equation system.

3.1 The Control Flow

In this section, we define the first part of the equation system that describes the control flow of the program. It is convenient to describe the recursive computation of this equation system by means of hardware circuit templates as listed in Figure 1. It is straightforward to extract from the circuit netlist the transition functions of the flipflops which then form our equation system for the control flow.

The circuits used in Figure 1 have boolean valued inputs *start*, *susp*, and *kill*, and boolean valued outputs *inst*, *insd* and *term*. The event and valued signals that occur in the program are collected in the additional input E (the environment). As the circuits given in Figure 1 do only compute the control flow, they will only read the values of E, but do not manipulate them (this is the task of the data flow).

The meaning of the other inputs and outputs is the following: *start* is used to start the execution of the program implemented by the circuit. *susp* is used to suspend the current computation, i.e. *susp* simply 'freezes' all flipflops of the circuit. The *kill* input is used to abort the current computation in that it simply resets all flipflops of the circuit. *inst* signals that the circuit is *instantaneous*, i.e., *inst* holds iff the computation would immediately terminate when it would be started with the current

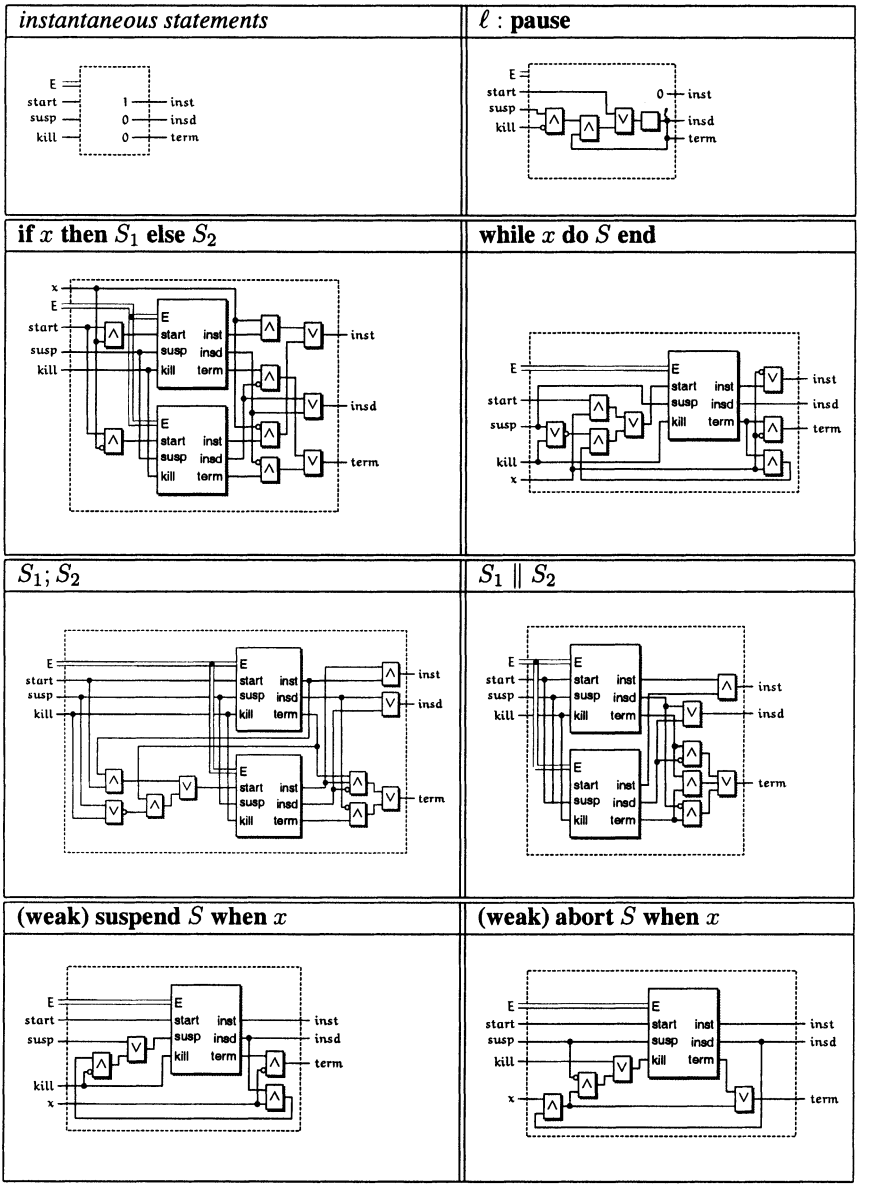

Figure 1 Hardware Circuits to Compute the Control Flow

environment E. $insd$ is the disjunction of all flipflops of the circuit, thus meaning that the control flow is currently somewhere inside the circuit. Finally, the output $term$ indicates that the current computation will now terminate (but the control flow is still in the circuit). If $term$ holds, and $start$ does not hold, the control flow will leave the circuit, so that $insd$ will be false at the next point of time. To avoid confusion, we denote in the following the $inst$, $insd$ and $term$ outputs of the circuit of a statement S as $\mathsf{inst}(S)$, $\mathsf{insd}(S)$, and $\mathsf{term}(S)$.

We have formally proved with the theorem prover HOL [14] that the recursive hardware synthesis rules as given in Figure 1 do correctly implement the control flow of a program. To be precise, the result is the following theorem (as S is equivalent to **suspend abort** S **when** 0 **when** 0, we can instantiate $kill := susp := 0$ to really compute the control flow of S):

Theorem 1 (Correctness of the Control Flow Computation) *For any* Quartz *statement S, the hardware circuit $\mathcal{C}(S)$ as generated by the rules given in Figure 1 implements the control flow of the statement* **suspend abort** S **when** $kill$ **when** $susp$, *provided that the following constraints do always hold:*

- $\neg(kill \wedge susp)$
- $insd \wedge \neg kill \wedge (\neg term \vee susp) \rightarrow \neg start$
- *For any loop body B of a substatement* **while** σ **do** B **end** *of S, the condition* $\mathsf{term}(B) \wedge \sigma \rightarrow \neg\mathsf{inst}(B)$ *must always hold.*

The first constraint means that circuits should not be both suspended or killed at any point of time. The second constraint roughly means that we must not start circuits that are already active ($insd$), unless they are not aborted ($\neg kill$) or terminate ($term$) at that point of time. The third constraint means that loop bodies must not be instantaneous. In fact, our constraint is a bit weaker, in that it allows loop bodies to be instantaneous if either σ does not hold, or the loop body does currently not terminate.

It is easily seen by the rules of Figure 1, that **pause** statements correspond with flipflops of the circuit. Given that $\ell_1, \ldots, \ell_p$ are the label variables of the **pause** statements that occur in a statement S, we can therefore derive an equation system of the form $\{\mathsf{init}(\ell_i) := 0 \mid 1 \leq i \leq p\} \cup \{\mathsf{next}(\ell_i) := \Omega_i \mid 1 \leq i \leq p\}$ from the circuit. The equations $\mathsf{init}(\ell_i) := 0$ thereby determine the initial state, and the equations $\mathsf{next}(\ell_i) := \Omega_i$ determine all transitions of the control flow. To distinguish the starting state from a possible termination state, we furthermore use an additional state variable ℓ_0 (the boot location) with the equations $\mathsf{init}(\ell_0) := 1$ and $\mathsf{next}(\ell_0) := 0$. We finally equate the $start$ input with ℓ_0 and the $kill$ and $susp$ inputs with 0.

3.2 *Defining the Data Flow*

We will now define the data flow part of the equation system. This is based on the set of guarded commands of S. In general, a guarded command is of the form (γ, c), where γ is a condition and c is a data manipulating statements, i.e., a statement of one of the following types: **emit** x, **emit delayed** x, $y := \tau$, or $y :=$ **delayed** τ. The

intuition behind a guarded command (γ, c) is that whenever the guard condition γ is satisfied, then the command c must be immediately executed. Guarded commands may themselves be viewed as a programming language (like Unity [15]) when we assume that each guarded command is a separate process. The set of guarded commands of a statement is computed as follows:

Definition 2 (Guarded Commands of Statements) *Given any* Quartz *statement* S, *we define the guarded commands* $\mathsf{guardcmd}(\varphi, S)$ *of* S *wrt. the initial condition* φ *as:*

- $\mathsf{guardcmd}(\varphi, \textbf{nothing}) \equiv \{\}$
- $\mathsf{guardcmd}(\varphi, \ell : \textbf{pause}) \equiv \{\}$
- $\mathsf{guardcmd}(\varphi, \textbf{emit}\ x) \equiv \{(\varphi, \textbf{emit}\ x)\}$
- $\mathsf{guardcmd}(\varphi, \textbf{emit delayed}\ x) \equiv \{(\varphi, \textbf{emit delayed}\ x)\}$
- $\mathsf{guardcmd}(\varphi, x := \tau) \equiv \{(\varphi, x := \tau)\}$
- $\mathsf{guardcmd}(\varphi, x := \textbf{delayed}\ \tau) \equiv \{(\varphi, x := \textbf{delayed}\ \tau)\}$
- $\mathsf{guardcmd}(\varphi, \textbf{if}\ \sigma\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})$
  $\equiv \mathsf{guardcmd}(\varphi \wedge \sigma, S_1) \cup \mathsf{guardcmd}(\varphi \wedge \neg\sigma, S_2)$
- $\mathsf{guardcmd}(\varphi, S_1; S_2)$
  $\equiv \mathsf{guardcmd}(\varphi, S_1) \cup \mathsf{guardcmd}(\mathsf{inst}(S_1) \wedge \varphi \vee \mathsf{term}(S_1), S_2)$
- $\mathsf{guardcmd}(\varphi, S_1 \parallel S_2) \equiv \mathsf{guardcmd}(\varphi, S_1) \cup \mathsf{guardcmd}(\varphi, S_2)$
- $\mathsf{guardcmd}(\varphi, \textbf{while}\ \sigma\ \textbf{do}\ S\ \textbf{end}) \equiv \mathsf{guardcmd}((\varphi \vee \mathsf{term}(S)) \wedge \sigma, S)$
- $\mathsf{guardcmd}(\varphi, \textbf{suspend}\ S\ \textbf{when}\ \sigma)$
  $\equiv \{(\gamma \wedge (\mathsf{insd}(S) \rightarrow \neg\sigma), \alpha) \mid (\gamma, \alpha) \in \mathsf{guardcmd}(\varphi, S)\}$
- $\mathsf{guardcmd}(\varphi, \textbf{weak suspend}\ S\ \textbf{when}\ \sigma) \equiv \mathsf{guardcmd}(\varphi, S)$
- $\mathsf{guardcmd}(\varphi, \textbf{abort}\ S\ \textbf{when}\ \sigma)$
  $\equiv \{(\gamma \wedge (\mathsf{insd}(S) \rightarrow \neg\sigma), \alpha) \mid (\gamma, \alpha) \in \mathsf{guardcmd}(\varphi, S)\}$
- $\mathsf{guardcmd}(\varphi, \textbf{weak abort}\ S\ \textbf{when}\ \sigma) \equiv \mathsf{guardcmd}(\varphi, S)$

The above definition should be clear, we only explain the case for a sequence $S_1; S_2$. The first part S_1, namely that $\mathsf{guardcmd}(\varphi, S_1; S_2)$ contains $\mathsf{guardcmd}(\varphi, S_1)$ is clear. For the second part, we have to distinguish between two cases: On the one hand S_1 may be instantaneous, so that the last location was the one described by φ. Hence, we have to compute $\mathsf{guardcmd}(\mathsf{inst}(S_1) \wedge \varphi, S_2)$. On the other hand, S_1 may not be instantaneous. In this case, we have to compute the last location inside S_1 where the control flow has been before leaving S_1. As this is encoded in $\mathsf{term}(S_1)$, we simply have to add $\mathsf{guardcmd}(\mathsf{term}(S_1), S_2)$.

Note that the weak and strong variants of suspension and abortion differ in that the strong variants replace the guards γ of S by $\gamma \wedge (\mathsf{insd}(S) \rightarrow \neg\sigma)$, so that no data manipulation takes place at abortion/suspension time.

For the definition of the data flow, we have to take into account that event and valued signals are handled differently. Remember that the values of event signals must be computed anew at each point of time, whereas the values of valued signals are stored unless an assignment changes them. A further problem of Quartz is here

that we must also cope with delayed emissions and assignments. Hence, we give the following definition (that does however only hold in case we have no write conflicts).

Definition 3 (Data Flow of Statements) *Assume the guarded commands* $(\alpha_1,$ **emit** $x)$, $\ldots$, $(\alpha_m,$ **emit** $x)$ *and* $(\beta_1,$ **emit delayed** $x)$, $\ldots$, $(\beta_n,$ **emit delayed** $x)$ *are the only emissions of the event signal* x *in a statement* S *for the initial condition* ℓ_0*. Then, we define:*

- $\mathcal{I}_{\mathsf{df}}(\ell_0, x, S) \equiv \{\mathsf{init}(x) := \bigvee_{i=1}^{m} \alpha_i\}$
- $\mathcal{R}_{\mathsf{df}}(\ell_0, x, S) \equiv \{\mathsf{next}(x) := (\bigvee_{i=1}^{n} \beta_i) \vee \mathsf{next}(\bigvee_{i=1}^{m} \alpha_i)\}$

Further, assume the guarded commands $(\alpha_1, y := \tau_1)$, $\ldots$, $(\alpha_m, y := \tau_m)$ *and* $(\beta_1, y :=$ **delayed** $\pi_1)$, $\ldots$, $(\beta_n, y :=$ **delayed** $\pi_n)$ *are the only assignments to the valued signal* y *in a statement* S *for the initial condition* ℓ_0*. Then, we define:*

- $$\mathcal{I}_{\mathsf{df}}(\ell_0, y, S) \equiv \left\{ \mathsf{init}(y) := \left(\begin{array}{l} \mathsf{if}\ \alpha_1\ \mathsf{then}\ \tau_1 \\ \vdots \\ \mathsf{elsif}\ \alpha_m\ \mathsf{then}\ \tau_n \\ \mathsf{else}\ \text{`some\_initial\_value'} \end{array} \right) \right\}$$
- $$\mathcal{R}_{\mathsf{df}}(\ell_0, y, S) \equiv \left\{ \mathsf{next}(y) := \left(\begin{array}{l} \mathsf{if}\ \mathsf{next}(\alpha_1)\ \mathsf{then}\ \mathsf{next}(\tau_1) \\ \mathsf{elsif}\ \mathsf{next}(\alpha_2)\ \mathsf{then}\ \mathsf{next}(\tau_2) \\ \vdots \\ \mathsf{elsif}\ \mathsf{next}(\alpha_m)\ \mathsf{then}\ \mathsf{next}(\tau_m) \\ \mathsf{elsif}\ \beta_1\ \mathsf{then}\ \pi_1 \\ \vdots \\ \mathsf{elsif}\ \beta_n\ \mathsf{then}\ \pi_n \\ \mathsf{else}\ y \end{array} \right) \right\}$$

Given a statement S *with the outputs* $x_1, \ldots, x_m$*, we define the data flow of* S *as the following equation system:*

- $\mathcal{I}_{\mathsf{df}}(\ell_0, S) \equiv \bigcup_{i=1}^{n} \mathcal{I}_{\mathsf{df}}(\ell_0, x_i, S)$
- $\mathcal{R}_{\mathsf{df}}(\ell_0, S) \equiv \bigcup_{i=1}^{n} \mathcal{R}_{\mathsf{df}}(\ell_0, x_i, S)$

3.3 Combining Control and Data Flow

Having defined the control flow and the data flow equation systems, it is now easy to combine both to obtain a complete description of the semantics of a statement. This is simply defined as given below:

Definition 4 (Equation System of a Statement) *Given a statement* S*, with an initial location* ℓ_0*. Then, we define the equation system for* ℓ_0 *and* S *as follows:*

- $\mathcal{I}(\ell_0, S) \equiv \mathcal{I}_{\mathsf{cf}}(\ell_0, S) \cup \mathcal{I}_{\mathsf{df}}(\ell_0, S)$
- $\mathcal{R}(\ell_0, S) \equiv \mathcal{R}_{\mathsf{cf}}(\ell_0, S) \cup \mathcal{R}_{\mathsf{df}}(\ell_0, S)$

4 APPLICATIONS AND SUMMARY

To summarize, we have developed a new translation of Esterel/Quartz programs into equivalent equation systems. These equivalent equation systems can be used for hardware and software synthesis, but also for a formal verification, which is our main focus. In particular, the translation to equation systems offers beneath the use of model checking techniques [13] also the use of term rewrite techniques as available in some theorem provers like HOL. Our translation does not suffer from schizophrenia problems in the control flow (schizophrenic synchronizers) [12]. The reason for this is that our circuit templates trigger themselves, i.e., we do not need an additional 'resume' input. This makes the entire hardware synthesis clearer and even more efficient: a simple comparison shows that our circuit templates require less gates than previous translations [12]. We have moreover proved the correctness of our translation with the HOL theorem prover, so that the theorem prover can even be used to implement a formal synthesis tool.

References

[1] Esterel Web. http://www.esterel.org.

[2] Simulog . http://www.simulog.fr.

[3] Cadence Design Systems,Inc. http://www.cadence.com.

[4] ECL Homepage. http://www-cad.eecs.berkeley.edu/

[5] Jester Homepage. http://www.parades.rm.cnr.it/projects/jester/jester.html.

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

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

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, 1991.

[9] P. Le Guernic, T. Gauthier, M. Le Borgne, and C. Le Maire. Programming real-time applications with SIGNAL. *IEEE*, 79(9):1321–1336, 1991.

[10] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing*, pp. 231–274, 1987.

[11] Ch. Andre. Syncharts: A visual representation of reactive behaviors. research report tr95-52, University of Nice, Sophia Antipolis, 1995.

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

[13] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *IEEE Symposium on Logic in Computer Science*, pp. 1–33, Washington, June 1990. IEEE Computer Society Press.

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

[15] K.M. Chandry and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.