# Embedding Imperative Synchronous Languages in Interactive Theorem Provers

K. Schneider
Universität Karlsruhe
Institut für Rechnerentwurf und Fehlertoleranz (Prof. Dr.-Ing. D. Schmid)
P.O. Box 6980, 76128 Karlsruhe, Germany
E-mail: Klaus.Schneider@informatik.uni-karlsruhe.de
http://goethe.ira.uka.de/~schneider

## Abstract

*We present a new way to define the semantics of imperative synchronous languages by means of separating the control and the data flow. The control flow is defined by predicates that describe entering conditions, conditions for internal moves, and termination conditions. The data flow is based on the extraction of guarded commands. This definition principle can be applied to any imperative synchronous language like* Esterel *or some statechart variants. Following this definition principle, we have embedded our language* Quartz *(an* Esterel *variant) in the interactive theorem prover* HOL. *We use this embedding for formal verification (both interactive theorem proving and symbolic model checking), program analysis, reasoning about the language at a meta-level, and verified code generation (formal synthesis).*

## 1. Introduction

Synchronous languages are becoming more and more attractive [15, 10, 13, 20] for the design and the verification of reactive real time systems. There are imperative languages like Esterel [6, 3, 5, 15], data flow languages like Lustre [18], and graphical languages like some Statechart [19] variants as SyncCharts [1]. We concentrate in this paper on imperative synchronous languages, but note that graphical and imperative synchronous languages can be naturally translated into each other [1].

The basic paradigm of these languages is the *perfect synchrony*, which means that most of the statements are executed as 'microsteps' in zero time. Consumption of time, i.e., the beginning of a new 'macrostep', must be explicitly programmed with special statements like the **pause** statement in Esterel: The execution of a **pause** statement consumes one logical unit of time, and this statement is the only one of Esterel's basic statements that consumes time at all. Consequently, all threads of a synchronous program run in lockstep: they execute the code between two **pause** statements in zero time, and automatically synchronize at the next **pause** statement.

As the control flow of a synchronous program $\mathcal{P}$ can therefore only rest at **pause** statements, it follows that the control flow of $\mathcal{P}$ can be represented by a finite state machine $\mathcal{A}_\mathcal{P}$: the states of $\mathcal{A}_\mathcal{P}$ are the possible control points of the program, i.e., points in the program text, where the control flow might rest for one unit of time. As the language allows the implementation of parallel threads, there might be more than one current position of the control flow in the program. A transition between two control states is enabled if some condition on the data values is satisfied. Execution of a transition will then invoke some manipulations of the data values. Hence, the semantics can be represented by a finite state control flow that interacts with a data flow of finitely many variables of possibly infinite data types.

A lot of different ways have already been studied to define the semantics of imperative synchronous languages: [4] distinguishes between semantics based on process algebras (in the structure-of-semantics style due to Plotkin), finite state machines, and another one that is directly based on hardware circuits.

In this paper, we present a new way to define the semantics of imperative synchronous languages like Esterel. For this purpose, we consider our own synchronous language called Quartz [25] that is a variant of Esterel, but want to emphasize that the way we embed the language can be applied to any other imperative synchronous language as well. *Our main interest is thereby that our semantics allows an easy* embedding *in an interactive theorem prover.* 'Embedding' thereby means that the set of programs will be formally defined as a type of the logic of the theorem prover, so that programs themselves become formulas of the logic. Based on such an embedding, one is able to formally reason about the language at a meta-level, and also to reason about particular programs. For example, we have already proved the correctness of a hardware synthesis [25] that further-

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

more allows us to use the theorem prover for a verified code generation.

A crucial problem for embedding languages (or more general theories) in already existing logics is to avoid inconsistencies: Simply postulating a set of axioms may lead to inconsistent theories so that everything could be derived. State-of-the-art theorem provers like HOL [17] therefore use certain definition principles to preserve the consistency. One main definition principle that guarantees such a conservative extension is primitive recursion [22, 23]. Primitive recursive definitions have moreover the advantage that the theorem prover can immediately derive suitable induction rules for interactive or automatic reasoning.

However, it is not straightforward to define the semantics of a language by means of primitive recursion. In particular, the process-algebraic semantics of Esterel does not allow such a definition, since the rule for loops recursively calls itself without 'decreasing' the program (with respect to some well-founded ordering). We therefore developed a new way to define the semantics such that only primitive recursion on the type of Quartz statements was necessary for all definitions.

One key to our semantics is the separation between control and data flow, which is a well-known technique for hardware designers. The definition of the control flow (Section 3.1) is based on control flow predicates $\mathsf{enter}(S)$, $\mathsf{move}(S)$, and $\mathsf{term}(S)$, that describe entering conditions, conditions for internal moves, and termination conditions of a statement $S$, respectively. The data flow of a statement is defined by its *guarded commands* (Section 3.2). These are of the form $(\gamma, \mathcal{C})$, and mean that whenever $\gamma$ holds, we execute the data manipulating statement $\mathcal{C}$.

Using our primitive recursive definitions, we have embedded our language Quartz in the interactive theorem prover HOL [17]. We present the definitions more or less as they appear in the theorem prover. We use our embedding for various purposes as e.g., formal synthesis [25], reasoning about the language Quartz, and of course, for the formal verification of program properties.

Please note that our language Quartz differs from Esterel in several points: We found it important to extend the Quartz language with delayed data assignments since this is easily allows us to describe many (sequential) algorithms and also hardware circuits: any register works with a delay of one cycle, and this is also the case for some Statechart languages. Moreover, our valued variables do not carry an additional status, since this if often unwanted and makes symbolic model checking less efficient. Finally, we only use one sort of conditional statement, in contrast to Esterel's 'if-then-else' and 'present-then-else' statements, and we do currently not support Esterel's variables that may change withing microsteps rather than in macrosteps only. In addition to Esterel, we also consider nondeterministic behavior and asynchronous concurrency.

The paper is organized as follows: in the next two sections, we define the syntax and semantics of Quartz, respectively. In Section 4, we then briefly list some experiences with our embedding. The paper then ends with some conclusions.

## 2. Syntax and Informal Semantics

The presentation of the language Quartz and its semantics should be done in the usual way, i.e., we should first define the available types, the expressions, the statements, and finally the programs. As Quartz statements mainly concentrate on the control flow of concurrent programs, we do neither consider particular types nor particular expressions in the following, and simply assume that we have some expressions over a type $\alpha$ (the HOL implementation used a polymorphic type $\alpha$ so that we may consider the HOL logic as a host language that provides us with the expressions and types).

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 $\alpha$. In general, we distinguish 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 data flow of these two kinds of variables is quite different: 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 at the next point of time. On the other hand, the value of an event variable $x$ is not sticky: when time elapses, the value of $x$ is reset to 0 (we denote Boolean values as 1 and 0), if it is not explicitly made 1 at the considered point of time. 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.

Event variables are made present with the **emit** statement, while the values of state variables are manipulated with usual assignments (**:=**). Of course, any event or state variable may also be an input variable, so that their values are determined by the environment only. **emit** statements and assignments **:=** are all data manipulating statements. The remaining basic statement 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*, $\ell$ is a location variable, $x$ is an event variable, $y$ is a state variable, and $\sigma$ is a Boolean expression:*

- **nothing** *(empty statement)*
- **emit** $x$ *and* **emit delayed** $x$ *(emissions)*

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

- $y := \tau$ *and* $y :=$ **delayed** $\tau$ *(assignments)*
- $\ell$ : **pause** *(consumption of time)*
- **if** $\sigma$ **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)*
- **while** $\sigma$ **do** $S$ **end** *(iteration)*
- **suspend** $S$ **when** $\sigma$ *(suspension)*
- **weak suspend** $S$ **when** $\sigma$ *(weak suspension)*
- **abort** $S$ **when** $\sigma$ *(abortion)*
- **weak abort** $S$ **when** $\sigma$ *(weak abortion)*
- **local** $x$ **in** $S$ **end** *(local event variable)*
- **local** $y : \alpha$ **in** $S$ **end** *(local state variable)*
- **now** $\sigma$ *(instantaneous assertion)*
- **during** $S$ **holds** $\sigma$ *(invariant assertion)*
- **run** $m(\tau_1, \ldots, \tau_n)$ *(running modules)*

Before giving a precise formal semantics, we informally discuss the meaning of the above statements (for further explanations and examples, we refer to [5]). 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*.

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. Hence, **nothing** is an instantaneous statement. Executing **emit** $x$ makes the event variable $x$ immediately present, i.e., the value of $x$ at that point of time is then 1. Executing an assignment $y := \tau$ will immediately change the value of $y$ to the current value of the expression $\tau$. The statements **emit delayed** $x$ and $y :=$ **delayed** $\tau$ are similarly defined as **emit** $x$ and $y := \tau$, respectively, but with a delay of one unit of time. In the latter statement, $\tau$ 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, but just consumes one logical unit of time. **pause** statements are endowed with location variables $\ell$ that we will use later on as a state variables to encode the control flow automaton. Of course, location variables must be unique for all occurrences.

**if** $\sigma$ **then** $S_1$ **else** $S_2$ **end** is a conditional statement: it immediately checks whether $\sigma$ evaluates to 1 or 0, and then immediately either executes $S_1$ or $S_2$ (depending on the value of $\sigma$). $S_1; S_2$ 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 enter both $S_1$ and $S_2$ and proceed with the execution of both $S_1$ and $S_2$. As long as both $S_1$ and $S_2$ are active, both threads are synchronously executed in lockstep. If $S_1$ terminates, but $S_2$ does not terminate, then $S_1 \parallel S_2$ behaves further as $S_2$ does (and vice versa). If finally $S_2$ terminates, then so does $S_1 \parallel S_2$. Beneath the synchronous parallel execution, Quartz offers also the asynchronous parallel execution $S_1 \parallel\!\!\parallel S_2$ of statements $S_1$ and $S_2$. The difference is that one of the threads may execute more than one macrostep while the other one only 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 $S_1 \parallel\!\!\parallel S_2$ by standard Esterel statements using additional inputs (cf. section 3.4).

Another Quartz statement that does not belong to Esterel is the nondeterministic choice: **choose** $S_1 \,[]\, S_2$ **end** will nondeterministically either execute $S_1$ or $S_2$. There is no need for such a statement as long as we want to write programs that are further translated to software or hardware. However, when reactive systems are *modeled*, e.g., for verification, there is often a requirement to hide certain details, and this will naturally yield nondeterministic systems. Nondeterministic systems may also be useful in early design stages where some implementation details are not yet fixed.

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

**(weak) suspend** $S$ **when** $\sigma$ implements process suspension, i.e. $S$ is entered when the execution of this statement starts (regardless of the current value of $\sigma$). For the following points of time, however, the execution of $S$ only proceeds if $\sigma$ evaluates to 0, otherwise its execution is 'frozen' until $\sigma$ releases the further execution.

Beneath suspension, abortion of processes is an important task for the process management. This is realized with the **abort** $S$ **when** $\sigma$ statement: $S$ is immediately entered at starting time (regardless of the current value of $\sigma$). $S$ is then executed as long as $\sigma$ is 0. If $\sigma$ becomes 1 during the execution of $S$, then $S$ is aborted. Hence, **abort** $S$ **when** $\sigma$ can 'normally' terminate, i.e., when the execution of $S$ terminates, or it may terminate by process abortion when $\sigma$ enforces this.

The 'weak' variants of process suspension and abortion differ on the data manipulations at suspension or abortion

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

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 do not ignore the value of the condition $\sigma$ at starting time (cf. section 3.4).

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 that there is no shadowing of variables, i.e., that all local variable names are different from each other and also different from all input and output variables.

Quartz does also allow us to write down assertions that must hold when the control flow reaches a certain point. Instantaneous assertions are given by the **now** statements: The meaning of **now** $\sigma$ is that $\sigma$ must hold at this point of time; if $\sigma$ would not hold, the execution will immediately stop (in a deadend state). Moreover, an assertion can also be required to hold *during* the execution of a statement $S$ using the statement **during** $S$ **holds** $\sigma$. It behaves like $S$, but additionally demands that whenever the control flow is inside $S$, then the condition $\sigma$ must hold.

Finally, the **run** statement is used to call already existing Quartz modules (see below). There are also a lot of other convenient statements that can be defined as macro expansions of basic statements (cf. Section 3.4 and [5]).

Similar to Esterel, Quartz allows us to define modules so that systems can be hierarchically organized. Moreover, these parts can be reused, which means that they can be multiply instantiated within another statement using the **run** statement. In general, a Quartz module is of the following form:

**Definition 2 (**Quartz **Modules)** *Given a basic* Quartz *statement $S$ with the event variables $a_1, \ldots, a_n, x_1, \ldots, x_p$ and the state variables $b_1, \ldots, b_m$, and $y_1, \ldots, y_q$, such that $S$ contains no emission of a variable $a_1, \ldots, a_n$ and no assignment to a variable $b_1, \ldots, b_m$. Moreover, assume that $c_1, \ldots, c_r$ are some further Boolean variables. Then, the following is a* Quartz *module:*

**module** $m$
  **input** $a_1, \ldots, a_n, b_1 : \alpha_1, \ldots, b_m : \alpha_m$;
  **output** $x_1, \ldots, x_p, y_1 : \beta_1, \ldots, y_q : \beta_q$;
  **control** $c_1, \ldots, c_r$;
  $S$
**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 furthermore specifies their types ($\alpha_j$ and $\beta_j$). The control variables are used to implement asynchronous concurrency and nondeterminism in that these variables are used as additional inputs (that are however not visible in the interface for other modules).

## 3. The Formal Semantics

In this section, we define the formal semantics of Quartz statements in that we first define the control flow, then the data flow, and finally combine both. Due to several technical problems, we first neglect asynchronous concurrency, nondeterministic choice, and local variable declarations in the next subsections. Section 3.4 will then explain how the semantics of these statements is defined.

Throughout the paper, the Boolean operators are meant to work on time dependent Boolean values. We use the usual Boolean operations $\neg$, $\wedge$, and $\vee$ for negation, conjunction, and disjunction, respectively. The Boolean constants for true and false are denoted as 1 and 0, respectively. Moreover, for any expression $\varphi$, $\mathsf{X}\varphi$ denotes the value of $\varphi$ at the next point of time. Furthermore, to avoid confusion with our definitions and the assignment operator $:=$, we use the symbol $:\equiv$ for our definitions.

### 3.1. Defining the Control Flow

Recall that the control flow can only rest at the **pause** statements of a program. Hence, to define the control flow, we need to consider the movement of the control flow from certain **pause** statements at the current point of time to possibly other **pause** statements at the next instant of time. For this reason, we have labeled the **pause** statements with unique location variables $\ell$ that are also called the labels of the **pause** statements.

The control flow of a Quartz statement $S$ is therefore a finite state machine whose states are encoded by the state variables $\mathsf{labels}(S)$. We define for any statement $S$ the predicates $\mathsf{enter}(S)$, $\mathsf{move}(S)$, and $\mathsf{term}(S)$ that describe the entering conditions of $S$, the conditions of $S$ for internal moves, and the termination conditions of $S$, respectively. The transition relation of the control flow machine is then obtained by combining these conditions. We introduce the following abbreviations: for any statement $S$, the set $\mathsf{labels}(S)$ is the set of labels $\ell$ occurring in $S$, and

$$\mathsf{in}(S) :\equiv \bigvee_{\ell \in \mathsf{labels}(S)} \ell$$

denotes all situations where the control flow is currently somewhere inside $S$. As the first primitive recursive definition, we need to define the cases when a statement instantaneously terminates:

**Definition 3 (Instantaneous Statements)** *Given a* Quartz *statement $S$, the formula* $\mathsf{inst}(S)$ *describes all conditions*

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 ©2001 IEEE

*where $S$ instantaneously terminates.* $\mathsf{inst}(S)$ *is recursively defined as follows:*

- $\mathsf{inst}(\textbf{nothing}) :\equiv 1$
- $\mathsf{inst}(\textbf{emit}\ x) :\equiv \mathsf{inst}(\textbf{emit delayed}\ x) :\equiv 1$
- $\mathsf{inst}(x := \tau) :\equiv \mathsf{inst}(x := \textbf{delayed}\ \tau) :\equiv 1$
- $\mathsf{inst}(\ell : \textbf{pause}) :\equiv 0$
- $\mathsf{inst}(\textbf{if}\ \sigma\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})$
  $:\equiv \sigma \wedge \mathsf{inst}(S_1) \vee \neg\sigma \wedge \mathsf{inst}(S_2)$
- $\mathsf{inst}(S_1; S_2) :\equiv \mathsf{inst}(S_1) \wedge \mathsf{inst}(S_2)$
- $\mathsf{inst}(S_1 \parallel S_2) :\equiv \mathsf{inst}(S_1) \wedge \mathsf{inst}(S_2)$
- $\mathsf{inst}(\textbf{while}\ \sigma\ \textbf{do}\ S\ \textbf{end}) :\equiv \neg\sigma \vee \mathsf{inst}(S)$
- $\mathsf{inst}(\textbf{(weak) suspend}\ S\ \textbf{when}\ \sigma) :\equiv \mathsf{inst}(S)$
- $\mathsf{inst}(\textbf{(weak) abort}\ S\ \textbf{when}\ \sigma) :\equiv \mathsf{inst}(S)$
- $\mathsf{inst}(\textbf{local}\ x\ \textbf{in}\ S\ \textbf{end})$
  $:\equiv \mathsf{inst}(\textbf{local}\ y : \alpha\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{inst}(S)$
- $\mathsf{inst}(\textbf{now}\ \sigma) :\equiv 1$
- $\mathsf{inst}(\textbf{during}\ S\ \textbf{holds}\ \sigma) :\equiv \mathsf{inst}(S)$

Note that one and the same statement can be instantaneous for a certain input/output combination and may consume time for another input/output combination. For example, $\mathsf{inst}(\textbf{if}\ i\ \textbf{then}\ \ell : \textbf{pause; emit}\ y\ \textbf{else emit}\ x\ \textbf{end}) = \neg i$. Using $\mathsf{inst}(S)$, we define $\mathsf{enter}(S)$, which describes the entering conditions of the statement $S$.

**Definition 4 (Entering Statements)** *Given a* Quartz *statement $S$, the formula* $\mathsf{enter}(S)$ *describes all conditions where the control flow enters $S$.* $\mathsf{enter}(S)$ *is recursively defined as follows:*

- $\mathsf{enter}(\textbf{nothing}) :\equiv 0$
- $\mathsf{enter}(\textbf{emit}\ x) :\equiv \mathsf{enter}(\textbf{emit delayed}\ x) :\equiv 0$
- $\mathsf{enter}(x := \tau) :\equiv \mathsf{enter}(x := \textbf{delayed}\ \tau) :\equiv 0$
- $\mathsf{enter}(\ell : \textbf{pause}) :\equiv \mathsf{X}\ell$
- $\mathsf{enter}(\textbf{if}\ \sigma\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})$
  $:\equiv \begin{pmatrix} \mathsf{enter}(S_1) \wedge \neg\mathsf{Xin}(S_2) \wedge \sigma \vee \\ \mathsf{enter}(S_2) \wedge \neg\mathsf{Xin}(S_1) \wedge \neg\sigma \end{pmatrix}$
- $\mathsf{enter}(S_1; S_2)$
  $:\equiv \begin{pmatrix} \mathsf{enter}(S_1) \wedge \neg\mathsf{Xin}(S_2) \vee \\ \mathsf{enter}(S_2) \wedge \neg\mathsf{Xin}(S_1) \wedge \mathsf{inst}(S_1) \end{pmatrix}$
- $\mathsf{enter}(S_1 \parallel S_2)$
  $:\equiv \begin{pmatrix} \mathsf{enter}(S_2) \wedge \mathsf{inst}(S_1) \wedge \neg\mathsf{Xin}(S_1) \vee \\ \mathsf{enter}(S_1) \wedge \mathsf{inst}(S_2) \wedge \neg\mathsf{Xin}(S_2) \vee \\ \mathsf{enter}(S_1) \wedge \mathsf{enter}(S_2) \end{pmatrix}$
- $\mathsf{enter}(\textbf{while}\ \sigma\ \textbf{do}\ S\ \textbf{end}) :\equiv \sigma \wedge \mathsf{enter}(S)$
- $\mathsf{enter}(\textbf{suspend}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \mathsf{enter}(\textbf{weak suspend}\ S\ \textbf{when}\ \sigma) :\equiv \mathsf{enter}(S)$
- $\mathsf{enter}(\textbf{abort}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \mathsf{enter}(\textbf{weak abort}\ S\ \textbf{when}\ \sigma) :\equiv \mathsf{enter}(S)$
- $\mathsf{enter}(\textbf{local}\ x\ \textbf{in}\ S\ \textbf{end})$
  $:\equiv \mathsf{enter}(\textbf{local}\ y : \alpha\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{enter}(S)$
- $\mathsf{enter}(\textbf{now}\ \sigma) :\equiv 0$
- $\mathsf{enter}(\textbf{during}\ S\ \textbf{holds}\ \sigma) :\equiv \mathsf{enter}(S)$

Note that if $S$ is instantaneous, then we can not enter $S$. Therefore, the entering condition for the instantaneous basic statements is 0. We proceed with defining $\mathsf{term}(S)$ that describes the termination conditions of $S$. For this reason, we assume that the control flow already rests somewhere inside $S$ and now leaves $S$ (but may reenter $S$ at the same time).

**Definition 5 (Termination of Statements)** *Given a* Quartz *statement $S$, the formula* $\mathsf{term}(S)$ *describes all termination conditions of $S$.* $\mathsf{term}(S)$ *is recursively defined as follows:*

- $\mathsf{term}(\textbf{nothing}) :\equiv 0$
- $\mathsf{term}(\textbf{emit}\ x) :\equiv \mathsf{term}(\textbf{emit delayed}\ x) :\equiv 0$
- $\mathsf{term}(x := \tau) :\equiv \mathsf{term}(x := \textbf{delayed}\ \tau) :\equiv 0$
- $\mathsf{term}(\ell : \textbf{pause}) :\equiv \ell$
- $\mathsf{term}(\textbf{if}\ \sigma\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})$
  $:\equiv \begin{pmatrix} \mathsf{term}(S_1) \wedge \neg\mathsf{in}(S_2) \vee \\ \mathsf{term}(S_2) \wedge \neg\mathsf{in}(S_1) \end{pmatrix}$
- $\mathsf{term}(S_1; S_2)$
  $:\equiv \begin{pmatrix} \mathsf{term}(S_1) \wedge \neg\mathsf{in}(S_2) \wedge \mathsf{inst}(S_2) \vee \\ \mathsf{term}(S_2) \wedge \neg\mathsf{in}(S_1) \end{pmatrix}$
- $\mathsf{term}(S_1 \parallel S_2)$
  $:\equiv \begin{pmatrix} \mathsf{term}(S_1) \wedge \neg\mathsf{in}(S_2) \vee \\ \mathsf{term}(S_2) \wedge \neg\mathsf{in}(S_1) \vee \\ \mathsf{term}(S_1) \wedge \mathsf{term}(S_2) \end{pmatrix}$
- $\mathsf{term}(\textbf{while}\ \sigma\ \textbf{do}\ S\ \textbf{end}) :\equiv \neg\sigma \wedge \mathsf{term}(S)$
- $\mathsf{term}(\textbf{weak suspend}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \mathsf{term}(\textbf{suspend}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \neg\sigma \wedge \mathsf{term}(S)$
- $\mathsf{term}(\textbf{weak abort}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \mathsf{term}(\textbf{abort}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \mathsf{in}(S) \wedge \sigma \vee \mathsf{term}(S)$
- $\mathsf{term}(\textbf{local}\ x\ \textbf{in}\ S\ \textbf{end})$
  $:\equiv \mathsf{term}(\textbf{local}\ y : \alpha\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{term}(S)$
- $\mathsf{term}(\textbf{now}\ \sigma) :\equiv 0$
- $\mathsf{term}(\textbf{during}\ S\ \textbf{holds}\ \sigma) :\equiv \mathsf{term}(S)$

Note that $\mathsf{term}(S)$ says nothing about the next location, which is not possible, since $S$ may or may not be reentered when $S$ terminates. The above defined formulas $\mathsf{inst}(S)$, $\mathsf{enter}(S)$, and $\mathsf{term}(S)$ are now combined to define the internal transitions $\mathsf{move}(S)$. For this reason, we use the abbreviation $\mathsf{stutter}(S) :\equiv \bigwedge_{\ell \in \mathsf{labels}(S)} (\ell = \mathsf{X}\ell)$ for stuttering states, i.e., the situations where the control flow remains the same at the next instant of time.

**Definition 6 (Internal Moves of Statements)** *Given a* Quartz *statement $S$, the formula* $\mathsf{move}(S)$ *describes all conditions where the control flow moves from somewhere inside $S$ to possibly another location inside $S$.* $\mathsf{move}(S)$ *is recursively defined as follows:*

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

- $\mathsf{move}\,(\textbf{nothing}) :\equiv 0$
- $\mathsf{move}\,(\textbf{emit}\ x) :\equiv \mathsf{move}\,(\textbf{emit delayed}\ x) :\equiv 0$
- $\mathsf{move}\,(x := \tau) :\equiv \mathsf{move}\,(x := \textbf{delayed}\ \tau) :\equiv 0$
- $\mathsf{move}\,(\ell : \textbf{pause}) :\equiv 0$
- $\mathsf{move}\,(\textbf{if}\ \sigma\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end})$
  $:\equiv \left(\begin{array}{l} \mathsf{move}\,(S_1) \wedge \neg\mathsf{in}\,(S_2) \wedge \neg\mathsf{Xin}\,(S_2) \vee \\ \mathsf{move}\,(S_2) \wedge \neg\mathsf{in}\,(S_1) \wedge \neg\mathsf{Xin}\,(S_1) \end{array}\right)$
- $\mathsf{move}\,(S_1; S_2)$
  $:\equiv \left(\begin{array}{l} \mathsf{move}\,(S_1) \wedge \neg\mathsf{in}\,(S_2) \wedge \neg\mathsf{Xin}\,(S_2) \vee \\ \mathsf{move}\,(S_2) \wedge \neg\mathsf{in}\,(S_1) \wedge \neg\mathsf{Xin}\,(S_1) \vee \\ \mathsf{term}\,(S_1) \wedge \neg\mathsf{Xin}\,(S_1) \wedge \neg\mathsf{in}\,(S_2) \wedge \mathsf{enter}\,(S_2) \end{array}\right)$
- $\mathsf{move}\,(S_1 \parallel S_2)$
  $:\equiv \left(\begin{array}{l} \mathsf{move}\,(S_1) \wedge \neg\mathsf{in}\,(S_2) \wedge \neg\mathsf{Xin}\,(S_2) \vee \\ \mathsf{move}\,(S_2) \wedge \neg\mathsf{in}\,(S_1) \wedge \neg\mathsf{Xin}\,(S_1) \vee \\ \mathsf{move}\,(S_1) \wedge \mathsf{move}\,(S_2) \vee \\ \mathsf{move}\,(S_1) \wedge \mathsf{term}\,(S_2) \wedge \neg\mathsf{Xin}\,(S_2) \vee \\ \mathsf{move}\,(S_2) \wedge \mathsf{term}\,(S_1) \wedge \neg\mathsf{Xin}\,(S_1) \end{array}\right)$
- $\mathsf{move}\,(\textbf{while}\ \sigma\ \textbf{do}\ S\ \textbf{end})$
  $:\equiv \left(\begin{array}{l} \mathsf{move}\,(S) \vee \mathsf{term}\,(S) \wedge \sigma \wedge \mathsf{enter}\,(S) \end{array}\right)$
- $\mathsf{move}\,(\textbf{suspend}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \mathsf{move}\,(\textbf{weak suspend}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \sigma \wedge \mathsf{in}\,(S) \wedge \mathsf{stutter}\,(S) \vee \neg\sigma \wedge \mathsf{move}\,(S)$
- $\mathsf{move}\,(\textbf{abort}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \mathsf{move}\,(\textbf{weak abort}\ S\ \textbf{when}\ \sigma)$
  $:\equiv \neg\sigma \wedge \mathsf{move}\,(S)$
- $\mathsf{move}\,(\textbf{local}\ x\ \textbf{in}\ S\ \textbf{end})$
  $:\equiv \mathsf{move}\,(\textbf{local}\ y : \alpha\ \textbf{in}\ S\ \textbf{end}) :\equiv \mathsf{move}\,(S)$
- $\mathsf{move}\,(\textbf{now}\ \sigma) :\equiv 0$
- $\mathsf{move}\,(\textbf{during}\ S\ \textbf{holds}\ \sigma) :\equiv \mathsf{move}\,(S)$

Note that we always require in the definition of $\mathsf{move}\,(S)$ that the control flow is currently somewhere inside $S$ and will remain somewhere inside $S$ at the next point of time. It is important that for the conditional and for the sequential composition $S_1; S_2$, we add constraints to assure that the control flow could not be both in $S_1$ and $S_2$. Otherwise, it would be possible that new threads could be randomly generated. The relations $\mathsf{enter}\,(S)$, $\mathsf{term}\,(S)$ and $\mathsf{move}\,(S)$ can now be used to define the control flow of a statement $S$ as follows:

**Definition 7 (Control Flow)** *Given a* Quartz *statement $S$, and a start variable $st$ that does not occur in $S$, we define the set of initial states $\mathcal{I}_{\mathsf{cf}}(st, S)$ and the transition relation $\mathcal{R}_{\mathsf{cf}}(st, S)$ of the control flow automaton as follows:*

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

The control flow automaton is therefore a finite state machine whose states are encoded by the state variables $\mathsf{labels}\,(S)$. Transitions are labeled by conditions that are all encoded in the transition relation $\mathcal{R}_{\mathsf{cf}}(st, S)$. The automaton has only a single initial state, namely the one encoded by $\mathcal{I}_{\mathsf{cf}}(st, S) :\equiv \neg\mathsf{in}\,(S)$. The automaton remains in this state until the start variable $st$ holds (third disjunct). If $st$ holds, there are two possibilities: Either $S$ can be instantaneous, which is described in the first disjunct of $\mathcal{R}_{\mathsf{cf}}(st, S)$, or the control flow can enter $S$ which is described in the second disjunct. Once inside $S$, we may follow internal transitions (fourth disjunct), or the control flow might leave $S$ (third disjunct). Note that the start variable $st$ is only respected when $\neg\mathsf{in}\,(S) \vee \mathsf{term}\,(S)$ holds. This means that an already active statement $S$ is not restarted unless it terminates.

Using the HOL theorem prover we have proved a couple of simple properties of the control flow predicates that have been defined above. These properties are important for any kind of formal reasoning about programs, in particular, they could be important as lemmas for automatic proof procedures. The most important of these properties are summarized in the following lemma.

**Lemma 1 (Properties of Control Flow Predicates)** *For any* Quartz *statement $S$, the following facts hold for the control flow predicates:*

- $\mathsf{enter}\,(S) \rightarrow \mathsf{Xin}\,(S)$
- $\mathsf{enter}\,(S) \rightarrow \neg\mathsf{inst}\,(S)$
- $\mathsf{term}\,(S) \rightarrow \mathsf{in}\,(S)$
- $\mathsf{move}\,(S) \rightarrow \mathsf{in}\,(S) \wedge \mathsf{Xin}\,(S)$
- $\mathsf{move}\,(S) \rightarrow \neg\mathsf{term}\,(S)$
- $\mathsf{stutter}\,(S) \rightarrow (\mathsf{in}\,(S) = \mathsf{Xin}\,(S))$
- $\neg\mathsf{in}\,(S) \rightarrow (\mathsf{stutter}\,(S) = \neg\mathsf{Xin}\,(S))$

The proofs of the above properties are all straightforwardly done by a simple induction over the Quartz statement $S$. Although, they are all easy to prove, they nevertheless give some more insight in the meaning of the control flow predicates.

The transition relation for the control flow has been given in a disjunctive form above. Using the above lemma, it is furthermore possible to convert the transition relation into a conjunctive form, which is for many applications more advantageous. For example, if $\mathcal{R}_{\mathsf{cf}}(st, S)$ appears as an assumption in a goal to be proved, then one can use simple logical inference rules (like `STRIP_TAC` in HOL) to split $\mathcal{R}_{\mathsf{cf}}(st, S)$ into a couple of smaller assumptions, if it is given in a conjunctive form. On the other hand, the disjunctive form has advantages for model checking, since it directly supports a disjunctive partitioning of the transition relation [8].

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01$10.00 ©2001 IEEE

**Lemma 2 (Conjunctive Form of Control Flow Transition Relation)** *For any* Quartz *statement $S$, the control flow can be alternatively defined with the following conjunctive transition relation:*

$$\begin{pmatrix} ((\neg \mathsf{in}(S) \vee \mathsf{term}(S)) \wedge st \wedge \mathsf{inst}(S) \rightarrow \neg \mathsf{Xin}(S)) \wedge \\ ((\neg \mathsf{in}(S) \vee \mathsf{term}(S)) \wedge st \wedge \neg \mathsf{inst}(S) \rightarrow \mathsf{enter}(S)) \wedge \\ ((\neg \mathsf{in}(S) \vee \mathsf{term}(S)) \wedge \neg st \rightarrow \neg \mathsf{Xin}(S)) \wedge \\ (\mathsf{in}(S) \wedge \neg \mathsf{term}(S) \rightarrow \mathsf{move}(S)) \end{pmatrix}$$

For the proof of the above lemma, we have to note that for conditionals and sequences, at most one substatement may be active. Hence, for $S \equiv$ **if** $\sigma$ **then** $S_1$ **else** $S_2$ **end** or $S \equiv S_1; S_2$, we can derive from the assumption $\mathsf{in}(S)$, *and* the fact that the transition relation $\mathcal{R}_{\mathsf{cf}}(st, S)$ always holds, that $\mathsf{in}(S_1) \neq \mathsf{in}(S_2)$ holds. Note that (in a propositional sense) the above formula is not equivalent to $\mathcal{R}_{\mathsf{cf}}(st, S)$, but both formulas nevertheless define the same transition system.

Using these facts, it is even possible to prove a recursive computation schema for the transition relation. The rules given in the following theorem can be used to construct the transition relation of a statement from the transition relations of its substatements. Note, however, that we still need the definitions of the control flow predicates $\mathsf{in}(S)$, $\mathsf{inst}(S)$, and $\mathsf{term}(S)$ in the following theorem. However, we could circumvent the computation of $\mathsf{move}(S)$ by the following computation:

**Theorem 1 (Recursive Definition of Control Flow)** *For any* Quartz *statement $S$, the transition relation $\mathcal{R}_{\mathsf{cf}}(st, S)$ can be recursively computed according to the following laws, provided that the assumption $st \rightarrow \neg \mathsf{in}(S) \vee \mathsf{term}(S)$ holds, i.e., that statements are only started when they are not active or are currently terminating:*

- $\mathcal{R}_{\mathsf{cf}}(st,$ **nothing**$) \Leftrightarrow 1$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **emit** $x) \Leftrightarrow \mathcal{R}_{\mathsf{cf}}(st,$ **emit delayed** $x) \Leftrightarrow 1$
- $\mathcal{R}_{\mathsf{cf}}(st, x := \tau) \Leftrightarrow \mathcal{R}_{\mathsf{cf}}(st, x :=$ **delayed** $\tau) \Leftrightarrow 1$
- $\mathcal{R}_{\mathsf{cf}}(st, \ell :$ **pause**$) \Leftrightarrow (\mathsf{X}\ell = st)$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **if** $\sigma$ **then** $S_1$ **else** $S_2$ **end**$)$
  $\Leftrightarrow \begin{pmatrix} \mathcal{R}_{\mathsf{cf}}(st \wedge \sigma, S_1) \wedge \\ \mathcal{R}_{\mathsf{cf}}(st \wedge \neg\sigma, S_2) \wedge \\ (\mathsf{in}(S_1) \rightarrow \neg \mathsf{in}(S_2)) \end{pmatrix}$
- $\mathcal{R}_{\mathsf{cf}}(st, S_1; S_2)$
  $\Leftrightarrow \begin{pmatrix} \mathcal{R}_{\mathsf{cf}}(st, S_1) \wedge \\ \mathcal{R}_{\mathsf{cf}}(st \wedge \mathsf{inst}(S_1) \vee \mathsf{term}(S_1), S_2) \wedge \\ (\mathsf{in}(S_1) \rightarrow \neg \mathsf{in}(S_2)) \end{pmatrix}$
- $\mathcal{R}_{\mathsf{cf}}(st, S_1 \parallel S_2) \Leftrightarrow (\mathcal{R}_{\mathsf{cf}}(st, S_1) \wedge \mathcal{R}_{\mathsf{cf}}(st, S_2))$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **while** $\sigma$ **do** $S$ **end**$)$
  $\Leftrightarrow \begin{pmatrix} \mathcal{R}_{\mathsf{cf}}(\sigma \wedge (st \vee \mathsf{term}(S)), S) \wedge \\ (\mathsf{term}(S) \wedge \sigma \rightarrow \neg \mathsf{inst}(S)) \end{pmatrix}$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **suspend** $S$ **when** $\sigma$**)**
  $\Leftrightarrow \mathcal{R}_{\mathsf{cf}}(st,$ **weak suspend** $S$ **when** $\sigma$**)**
  $\Leftrightarrow \begin{pmatrix} \mathcal{R}_{\mathsf{cf}}(st, S) \wedge (\mathsf{in}(S) \rightarrow \neg\sigma) \vee \\ (\mathsf{in}(S) \wedge \sigma \wedge \mathsf{stutter}(S)) \end{pmatrix}$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **abort** $S$ **when** $\sigma$**)**
  $\Leftrightarrow \mathcal{R}_{\mathsf{cf}}(st,$ **weak abort** $S$ **when** $\sigma$**)**
  $\Leftrightarrow \begin{pmatrix} \mathcal{R}_{\mathsf{cf}}(st, S) \wedge (\mathsf{in}(S) \wedge \mathsf{Xin}(S) \rightarrow \neg\sigma) \vee \\ \mathsf{in}(S) \wedge \sigma \wedge st \wedge \mathsf{inst}(S) \wedge \neg \mathsf{Xin}(S) \vee \\ \mathsf{in}(S) \wedge \sigma \wedge st \wedge \mathsf{enter}(S) \vee \\ \mathsf{in}(S) \wedge \sigma \wedge \neg st \wedge \neg \mathsf{Xin}(S) \end{pmatrix}$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **local** $x$ **in** $S$ **end**$) \Leftrightarrow \mathcal{R}_{\mathsf{cf}}(st, S)$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **local** $y : \alpha$ **in** $S$ **end**$) \Leftrightarrow \mathcal{R}_{\mathsf{cf}}(st, S)$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **now** $\sigma) \Leftrightarrow 1$
- $\mathcal{R}_{\mathsf{cf}}(st,$ **during** $S$ **holds** $\sigma) \Leftrightarrow \mathcal{R}_{\mathsf{cf}}(st, S)$

The restriction $st \rightarrow \neg \mathsf{in}(S) \vee \mathsf{term}(S)$ for the start variable $st$ is not a severe one and must hold for any reasonable setting.

### 3.2. Defining the Data Flow

We will now define the data flow of a statement $S$. This is based on the set of guarded commands of $S$, which are pairs of the following kind:

**Definition 8 (Guarded Commands)** *A guarded command is a pair $(\gamma, \mathcal{C})$, where $\gamma$ is a Boolean expression called the guard, and $\mathcal{C}$ is a* Quartz *statement of one of the following forms:* **emit** $x$, **emit delayed** $x$, $y := \tau$, $y :=$ **delayed** $\tau$, *and* **now** $\sigma$.

The intuition behind a guarded command $(\gamma, \mathcal{C})$ is that whenever the condition $\gamma$ is satisfied, then we immediately execute the command $\mathcal{C}$. In case of the **now** statement, we instead demand that at this point of time, the condition $\sigma$ must hold. Guarded commands may be viewed as a programming language like Unity [11, 12] in that the guarded commands run as separate processes in parallel.

The set of guarded commands of the remaining Quartz statements is computed as follows, where we use a so called precondition $\varphi$. The precondition $\varphi$ contains the information about the current control flow state and the current variable values that provoke the execution of the statement. Clearly, we initially start with the precondition $st$.

**Definition 9 (Guarded Commands of Statements)** *Given a* Quartz *statement $S$ without local variable declarations, nondeterministic choice, and asynchronous parallel execution, we define the set of its guarded commands* $\mathsf{gcmd}(\varphi, S)$ *of $S$ for a precondition $\varphi$ as follows:*

- $\mathsf{gcmd}(\varphi,$ **nothing**$) :\equiv \{\}$
- $\mathsf{gcmd}(\varphi,$ **emit** $x) :\equiv \{(\varphi,$ **emit** $x)\}$
- $\mathsf{gcmd}(\varphi,$ **emit delayed** $x) :\equiv \{(\varphi,$ **emit delayed** $x)\}$
- $\mathsf{gcmd}(\varphi, x := \tau) :\equiv \{(\varphi, x := \tau)\}$
- $\mathsf{gcmd}(\varphi, x :=$ **delayed** $\tau) :\equiv \{(\varphi, x :=$ **delayed** $\tau)\}$
- $\mathsf{gcmd}(\varphi, \ell :$ **pause**$) :\equiv \{\}$

- $\mathsf{gcmd}\,(\varphi, \textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$
  $:\equiv \mathsf{gcmd}\,(\varphi \wedge \sigma, S_1) \cup \mathsf{gcmd}\,(\varphi \wedge \neg\sigma, S_2)$
- $\mathsf{gcmd}\,(\varphi, S_1; S_2)$
  $:\equiv \mathsf{gcmd}\,(\varphi, S_1)$
  $\cup\, \mathsf{gcmd}\,(\varphi \wedge \mathsf{inst}\,(S_1) \vee \mathsf{term}\,(S_1), S_2)$
- $\mathsf{gcmd}\,(\varphi, S_1 \parallel S_2)$
  $:\equiv \mathsf{gcmd}\,(\varphi, S_1) \cup \mathsf{gcmd}\,(\varphi, S_2)$
- $\mathsf{gcmd}\,(\varphi, \textbf{while } \sigma \textbf{ do } S \textbf{ end})$
  $:\equiv \mathsf{gcmd}\,((\varphi \vee \mathsf{term}\,(S)) \wedge \sigma, S)$
- $\mathsf{gcmd}\,(\varphi, \textbf{suspend } S \textbf{ when } \sigma)$
  $:\equiv \{(\gamma \wedge (\mathsf{in}\,(S) \rightarrow \neg\sigma), \alpha) \mid (\gamma, \alpha) \in \mathsf{gcmd}\,(\varphi, S)\}$
- $\mathsf{gcmd}\,(\varphi, \textbf{abort } S \textbf{ when } \sigma)$
  $:\equiv \{(\gamma \wedge (\mathsf{in}\,(S) \rightarrow \neg\sigma), \alpha) \mid (\gamma, \alpha) \in \mathsf{gcmd}\,(\varphi, S)\}$
- $\mathsf{gcmd}\,(\varphi, \textbf{weak suspend } S \textbf{ when } \sigma)$
  $:\equiv \mathsf{gcmd}\,(\varphi, \textbf{weak abort } S \textbf{ when } \sigma)$
  $:\equiv \mathsf{gcmd}\,(\varphi, S)$
- $\mathsf{gcmd}\,(\varphi, \textbf{now } \sigma) :\equiv \{(\varphi, \textbf{now } \sigma)\}$
- $\mathsf{gcmd}\,(\varphi, \textbf{during } S \textbf{ holds } \sigma) :\equiv \{(\mathsf{in}\,(S), \textbf{now } \sigma)\}$

The above definition should be clear for most statements. We just consider the definition for a sequence $S_1; S_2$: Of course, the control flow first enters $S_1$, so that $\mathsf{gcmd}\,(\varphi, S_1)$ is to be computed. For the second part ($S_2$), we have to distinguish between two cases: On the one hand, $S_1$ may be instantaneous, so that the precondition to reach $S_2$ is $\varphi \wedge \mathsf{inst}\,(S_1)$. 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 by $\mathsf{term}\,(S_1)$, we furthermore have to add guards of $S_2$ with the precondition $\mathsf{term}\,(S_1)$.

It is easily seen by induction along the above definition that *the precondition $\varphi$ will always be a propositional formula that refers to the current control flow location and the current variable values*. In particular, it does neither refer to the next control flow location nor to the next inputs/outputs.

For the definition of the data flow, we have to take into account that event variables and state variables are handled differently. Recall that the values of event variables are 'not sticky', i.e. are computed anew at each point of time, whereas the values of state variables are stored unless an assignment changes them.

**Definition 10 (Data Flow of Event Variables)** *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 an event variable $x$ in a* Quartz *statement $S$ for the initial precondition $st$. Then, we define:*

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

For the definition of the semantics of the event variables, we have to take care about two things: (1) only output event variables should be present that have an emitter, and (2) all output event variables should be present that have an emitter (condition (1) is often called the 'coherency' principle [4]).

At the initial point of time, there is an emitter for $x$ iff one of the $\alpha_i$'s holds. This already explains our definition of the initial equation. For the remaining points of time, there are two possibilities for an emitter for $x$: on the one hand, one of the $\alpha_i$'s may hold, but on the other hand, one of the $\beta_i$'s may have been true at the previous point of time. This yields in the more complicated form of the transition equation of $x$.

We will shortly define also the data flow of state output variables in form of an initial and a transition equation. However, for an intermediate discussion about write conflicts, we first give the following intermediate definition of the data flow of these variables.

**Definition 11 (Data Flow of Flow of State Variables)** *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 state variable $y$ in a statement $S$ for the initial precondition $st$. Then, we define:*

- $\mathcal{I}_{\mathsf{df}}(st, y, S) :\equiv \bigwedge_{i=1}^{m} (\alpha_i \rightarrow [y = \tau_i])$
- $\mathcal{R}_{\mathsf{df}}(st, y, S)$
  $:\equiv \begin{pmatrix} (\bigwedge_{i=1}^{m} \mathsf{X}\,[\alpha_i \rightarrow (y = \tau_i)]) \wedge \\ (\bigwedge_{i=1}^{n} [\beta_i \rightarrow (\mathsf{X}y = \pi_i)]) \wedge \\ ([\bigwedge_{i=1}^{m} \neg\mathsf{X}\alpha_i \wedge \bigwedge_{i=1}^{n} \neg\beta_i] \rightarrow [\mathsf{X}y = y]) \end{pmatrix}$

Clearly, whenever $\alpha_i$ holds, we have to execute the command $y := \tau_i$, so that the equation $y = \tau_i$ must then immediately hold. This holds for the initial point of time as well as for the remaining ones. If, however, $\beta_i$ holds at a point of time, then we immediately execute the command $y :=$ **delayed** $\pi_i$, so that the equation $\mathsf{X}y = \pi_i$ must hold at this point of time, i.e., the value of $y$ at the next point of time is $\pi_i$ ($\pi_i$ is evaluated at the current point of time). Finally, if neither a guard $\alpha_i$ nor a guard $\beta_i$ holds, the value of $y$ must be stored, i.e., we then have the equation $\mathsf{X}y = y$.

The above definition directly formalizes our intuitive use of guarded commands. However, the above formulas may become false in certain locations for certain variable values, since different equations could be enabled at the same time so that different equations should then hold. In these situations, we have a *write conflict*. Note that we do not have to care about these write conflicts for the output event variables: if an event variable is emitted twice, it is still present, and if one thread emits a variable $x$ at some point of time, but another one does not, the variable will nevertheless be present. For the state variables, we must however face the problem that different threads assign different state

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

to a variable. It is not decidable whether a given program has a write conflict or not. Nevertheless, we can postulate the (undecidable) conditions when write conflicts appear:

**Definition 12 (Write Conflicts)** *Assume the guarded commands* $(\alpha_1, y := \tau_1)$, ..., $(\alpha_m, y := \tau_m)$ *and* $(\beta_1, y :=$ **delayed** $\pi_1)$, ..., $(\beta_n, y :=$ **delayed** $\pi_n)$ *are the only assignments to the state variable* $y$ *in a* Quartz *statement* $S$ *for the initial condition* $st$*. Then, the formula* $\mathsf{WC}(y, st, S)$ *defined as below covers all situations where different assignments are made for* $y$*:*

$$\mathsf{WC}(y, st, S) :\equiv \left( \begin{array}{l} \bigvee_{i=1}^{m} \bigvee_{j=1}^{m} (\alpha_i \wedge \alpha_j \wedge \neg[\tau_i = \tau_j]) \vee \\ \bigvee_{i=1}^{n} \bigvee_{j=1}^{n} \mathsf{X}(\beta_i \wedge \beta_j \wedge \neg[\pi_i = \pi_j]) \vee \\ \bigvee_{i=1}^{m} \bigvee_{j=1}^{n} (\mathsf{X}\alpha_i \wedge \beta_j \wedge \neg[\mathsf{X}\tau_i = \pi_j]) \end{array} \right)$$

Hence, a program is free of write conflicts, if we compute its control and data flow as shown above, and are then able to verify that the above write conflict situations never take place. For first order data types, this is however an undecidable problem. An alternative approach, that is still undecidable, but nevertheless potentially simpler, is to check the above write conflict situations statically, thus ignoring reachable and unreachable states of the program. A first order theorem prover is sufficient for that purpose, and in case the data types were finite, also tautology checking could be feasible.

Beneath the possible write conflicts, another problem has to be solved for a deterministic representation of the data flow: It could be the case that the initial value of a state variable $y$ is not determined, i.e., that the disjunction of the guards $\alpha_i$ is not true at initial time. In this case, different ways can be followed to define the semantics. On the one hand, the initial value can be an arbitrary value of the corresponding type, but it is also reasonable to define for any type a default initial value $\tau_0$ that is to be used in this case. We use the second suggestion in the following definition:

**Definition 13 (Data Flow of State Variables)** *Assume the guarded commands* $(\alpha_1, y := \tau_1)$, ..., $(\alpha_m, y := \tau_m)$ *and* $(\beta_1, y :=$ **delayed** $\pi_1)$, ..., $(\beta_n, y :=$ **delayed** $\pi_n)$ *are the only assignments to the state variable* $y$ *in a statement* $S$ *for the initial precondition* $st$*. Assume further, that there are no write conflicts for* $y$ *in* $S$*. Then, we define:*

- $\mathcal{I}_{\mathsf{df}}(st, y, S) \equiv \left( y = \left( \begin{array}{l} \mathsf{if}\ \alpha_1\ \mathsf{then}\ \tau_1 \\ \vdots \\ \mathsf{elsif}\ \alpha_m\ \mathsf{then}\ \tau_n \\ \mathsf{else}\ \tau_0 \end{array} \right) \right)$
- $\mathcal{R}_{\mathsf{df}}(st, y, S) \equiv \left( \mathsf{X}y = \left( \begin{array}{l} \mathsf{if}\ \mathsf{X}\alpha_1\ \mathsf{then}\ \mathsf{X}\tau_1 \\ \mathsf{elsif}\ \mathsf{X}\alpha_1\ \mathsf{then}\ \mathsf{X}\tau_1 \\ \vdots \\ \mathsf{elsif}\ \mathsf{X}\alpha_m\ \mathsf{then}\ \mathsf{X}\tau_m \\ \mathsf{elsif}\ \beta_1\ \mathsf{then}\ \pi_1 \\ \vdots \\ \mathsf{elsif}\ \beta_n\ \mathsf{then}\ \pi_n \\ \mathsf{else}\ y \end{array} \right) \right)$

Note that the above definition is well-defined, i.e., it does not depend on the order of the guarded commands, since the absence of write conflicts means that the conditions $\alpha_i$ and $\mathsf{X}\beta_i$ either exclude each other or do not lead to a contradiction. It is therefore easily seen that the above definition is equivalent to the previous one.

Our previous definitions did only define the data flow for one variable of a statement $S$. It is obvious how to extend this definition for all variables of a statement $S$ as shown in the next definition.

**Definition 14 (Data Flow of Statements)** *Given a* Quartz *statement* $S$ *with the output (event or state) variables* $y_1$, ..., $y_m$*. Let moreover be* $(\varphi_1, \mathbf{now}\ \sigma_1)$, ..., $(\varphi_p, \mathbf{now}\ \sigma_p)$ *are all guarded commands of* $S$ *with assertions for the initial precondition* $st$*. Then, we define the data flow of* $S$ *as follows, provided that there are no write conflicts for the state output variables:*

- $\mathcal{I}_{\mathsf{df}}(st, S) :\equiv \bigwedge_{i=1}^{m} \mathcal{I}_{\mathsf{df}}(st, y_i, S)$
- $\mathcal{R}_{\mathsf{df}}(st, S) :\equiv \bigwedge_{i=1}^{m} \mathcal{R}_{\mathsf{df}}(st, y_i, S) \wedge \bigwedge_{i=1}^{p} (\varphi_i \rightarrow \sigma_i)$

It may be the case that the variable $y_i$ occurs both on the left and right hand side of a data flow equation. In this case, one speaks about *causality* cycles that may also extend to more than one equation. We do not consider the issue of *causality analysis* in this paper, but note that this is an important stage for compiling synchronous languages.

### 3.3. Combining Control and Data Flow

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

**Definition 15 (Semantics of a Statement)** *Given a* Quartz *statement* $S$*, with the start variable* $st$*. Then, we define the semantics for* $st$ *and* $S$ *as follows:*

- $\mathcal{I}(st, S) :\equiv \mathcal{I}_{\mathsf{cf}}(st, S) \wedge \mathcal{I}_{\mathsf{df}}(st, S)$
- $\mathcal{R}(st, S) :\equiv \mathcal{R}_{\mathsf{cf}}(st, S) \wedge \mathcal{R}_{\mathsf{df}}(st, S)$

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 ©2001 IEEE

### 3.4. Further Statements

We have only considered basic statements for defining the semantics of Quartz in the previous sections. There are many macro statements whose semantics is then simply given by a corresponding macro expansion to basic statements [5]. However, we have also excluded in the previous sections the asynchronous concurrency, the nondeterministic choice, and local variable declarations. We will now explain how these statements are handled.

Local variable declarations can be ignored for the semantics provided all variables are uniquely renamed, and *reincarnation problems* [5] are avoided. The latter can be simply done by unrolling loop bodies [5], although there are better possibilities [5].

Asynchronous concurrency and nondeterministic choice are easily implemented by additional control variables. Control variables are treated as input variables, but neither the user nor the environment of a module is not allowed to connect them with outputs of other modules. Hence, values of control variables are nondeterministically generated by the environment (which is our source of nondeterminism).

Nondeterministic choice statements are reduced by the following equation to deterministic choices with a new control variable $c$:

$$\textbf{choose } S_1 \ [] \ S_2 \textbf{ end} :\equiv (\textbf{if } c \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$$

As inputs occur nondeterministically, it follows that due to the above replacement, we deterministically react to inputs that are nondeterministically generated by the environment. In a similar way, we reduce the asynchronous concurrency to a synchronous one by the following equation, where we need two new control variables $r_1$ and $r_2$:

$$S_1 \ ||| \ S_2 :\equiv \left( \begin{array}{l} \textbf{during} \\ \quad \textbf{suspend } S_1 \textbf{ when } \neg r_1 \\ \quad \| \\ \quad \textbf{suspend } S_2 \textbf{ when } \neg r_2 \\ \textbf{holds } [\mathsf{in}\,(S_1) \wedge r_1] \vee [\mathsf{in}\,(S_2) \wedge r_2] \end{array} \right)$$

For the elimination of the asynchronous parallel execution, we have added two control variables $r_1$ and $r_2$ that allow thread $S_1$ and $S_2$ to run, respectively: $S_i$ will proceed with its execution iff $r_i$ holds. The additional constraint will furthermore demand that at least one of the two threads will proceed with its execution when both are active. Note that under the assumption $\mathsf{in}\,(S_1) \vee \mathsf{in}\,(S_2)$, the constraint $[\mathsf{in}\,(S_1) \wedge r_1] \vee [\mathsf{in}\,(S_2) \wedge r_2]$ is equivalent to:

$$\left( \begin{array}{l} [\mathsf{in}\,(S_1) \wedge \mathsf{in}\,(S_2) \rightarrow (r_1 \vee r_2)] \wedge \\ [\mathsf{in}\,(S_1) \wedge \neg \mathsf{in}\,(S_2) \rightarrow r_1] \wedge \\ [\mathsf{in}\,(S_2) \wedge \neg \mathsf{in}\,(S_1) \rightarrow r_2] \end{array} \right)$$

In the same manner, we could also introduce other forms of concurrency, as e.g., *interleaving*: With the same replacement of the statement, we have to use the constraint $[\mathsf{in}\,(S_1) \wedge r_1] \oplus [\mathsf{in}\,(S_2) \wedge r_2]$ which is under the assumption $\mathsf{in}\,(S_1) \vee \mathsf{in}\,(S_2)$ equivalent to:

$$\left( \begin{array}{l} [\mathsf{in}\,(S_1) \wedge \mathsf{in}\,(S_2) \rightarrow (r_1 \oplus r_2)] \wedge \\ [\mathsf{in}\,(S_1) \wedge \neg \mathsf{in}\,(S_2) \rightarrow r_1] \wedge \\ [\mathsf{in}\,(S_2) \wedge \neg \mathsf{in}\,(S_1) \rightarrow r_2] \end{array} \right)$$

Note that we only need to have equivalence for the cases where $\mathsf{in}\,(S_1) \vee \mathsf{in}\,(S_2)$ holds, since in the other cases the statement is not active. We can view the constraint of the **during** statement as a *scheduler* that controls the execution of the different threads.

Finally, we show how to define the weak immediate forms of suspension and abortions. To this end, we have to define the part of a statement that is instantaneously executed:

**Definition 16 (Surface of a Statement)** *Given a* Quartz *statement $S$, we define the* Quartz *statement* $\mathsf{surface}\,(S)$ *recursively as follows:*

- $\mathsf{surface}\,(\textbf{nothing}) :\equiv \textbf{nothing}$
- $\mathsf{surface}\,(\textbf{emit } x) :\equiv \textbf{emit } x$
- $\mathsf{surface}\,(\textbf{emit delayed } x) :\equiv \textbf{emit delayed } x$
- $\mathsf{surface}\,(x := \tau) :\equiv x := \tau$
- $\mathsf{surface}\,(x := \textbf{delayed } \tau) :\equiv x := \textbf{delayed } \tau$
- $\mathsf{surface}\,(\ell : \textbf{pause}) :\equiv \textbf{nothing}$
- $\mathsf{surface}\,(\textbf{if } \sigma \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end})$
  $:\equiv \textbf{if } \sigma \textbf{ then } \mathsf{surface}\,(S_1) \textbf{ else } \mathsf{surface}\,(S_2) \textbf{ end}$
- $\mathsf{surface}\,(S_1; S_2)$
  $:\equiv \ \mathsf{surface}\,(S_1)\,;$
  $\quad \textbf{if } \mathsf{inst}\,(S_1) \textbf{ then } \mathsf{surface}\,(S_2) \textbf{ else nothing end}$
- $\mathsf{surface}\,(S_1 \parallel S_2) :\equiv \mathsf{surface}\,(S_1) \parallel \mathsf{surface}\,(S_2)$
- $\mathsf{surface}\,(\textbf{while } \sigma \textbf{ do } S \textbf{ end})$
  $:\equiv \textbf{if } \sigma \textbf{ then } \mathsf{surface}\,(S) \textbf{ else nothing end}$
- $\mathsf{surface}\,(\textbf{suspend } S \textbf{ when } \sigma)$
  $:\equiv \mathsf{surface}\,(\textbf{weak suspend } S \textbf{ when } \sigma)$
  $:\equiv \mathsf{surface}\,(S)$
- $\mathsf{surface}\,(\textbf{abort } S \textbf{ when } \sigma)$
  $:\equiv \mathsf{surface}\,(\textbf{weak abort } S \textbf{ when } \sigma)$
  $:\equiv \mathsf{surface}\,(S)$
- $\mathsf{surface}\,(\textbf{now } \sigma) :\equiv \textbf{now } \sigma$
- $\mathsf{surface}\,(\textbf{during } S \textbf{ holds } \sigma) :\equiv \mathsf{surface}\,(S)$

Using the above definition, we can now define the weak immediate versions as follows:

**Definition 17 (Immediate Abortion and Suspension)** *We define immediate abortion and suspension as follows:*

- $\textbf{abort } S \textbf{ when immediate } \sigma$
  $:\equiv \textbf{if } \sigma \textbf{ then nothing else abort } S \textbf{ when } \sigma$
- $\ell : \textbf{suspend } S \textbf{ when immediate } \sigma$
  $:\equiv \textbf{while } \sigma \textbf{ do } \ell : \textbf{pause end}; \textbf{suspend } S \textbf{ when } \sigma$

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

- **weak abort** $S$ **when immediate** $\sigma$
  $:\equiv \left( \begin{array}{l} \textbf{if } \sigma \textbf{ then } \mathsf{surface}\,(S) \\ \textbf{else weak abort } S \textbf{ when } \sigma \end{array} \right)$
- $\ell$ : **weak suspend** $S$ **when immediate** $\sigma$
  $:\equiv \left( \begin{array}{l} \textbf{if } \sigma \textbf{ then} \\ \quad \mathsf{surface}\,(S)\,; \\ \quad \textbf{while } \sigma \textbf{ do } \ell : \textbf{pause end} \\ \textbf{end}; \\ \textbf{weak suspend } S \textbf{ when } \sigma \end{array} \right)$

# 4. Results

Using the definitions of the previous section, we have embedded Quartz in the HOL theorem prover. We use this embedding for several applications:

## 4.1. Formal Synthesis

In general, formal synthesis [21] is the process to derive a lower level implementation from a higher level implementation, such that each transformation step must be proved within a formal proof calculus. As Quartz programs are now terms of the HOL logic, we can manipulate them by HOL's inference rules. In particular, we can use HOL's rewrite machinery to translate Quartz statements $S$ to their finite state machines $\mathcal{A}_S$. Note that HOL's rewrite machinery thus computes a formal proof for the equivalence of $S$ and $\mathcal{A}_S$. This is particularly useful when a high level of safety has to be assured: HOL's implementation [17] guarantees the absence of errors in the code generation.

For reasons of efficiency, we note that it is really necessary to share common subformulas to achieve an efficient translation. For example, we obtain the following translation times for the arbiter program given in [27] (time in seconds for $n$ processes; on a Pentium III with 450 MHz):

| $n$ | time | $n$ | time | $n$ | time | $n$ | time |
|---|---|---|---|---|---|---|---|
| 5 | 1.610 | 30 | 13.060 | 55 | 38.070 | 80 | 74.600 |
| 10 | 3.160 | 35 | 16.650 | 60 | 42.560 | 85 | 87.080 |
| 15 | 5.040 | 40 | 20.590 | 65 | 49.360 | 90 | 96.060 |
| 20 | 7.870 | 45 | 25.340 | 70 | 59.110 | 95 | 107.200 |
| 25 | 10.060 | 50 | 30.450 | 75 | 64.930 | 100 | 123.170 |

## 4.2. Reasoning about Quartz

As we have implemented the new HOL type of Quartz programs, we can now write down formulas such as $\forall P, Q : \mathsf{Quartz}(\alpha).(P \parallel Q) = (Q \parallel P)$. Hence, we can formally argue about *all* Quartz programs and prove theorems about the language. In particular, we have proved *the determinism of a program whenever there are no data conflicts!* We have moreover proved the correctness of the hardware circuit synthesis as given in [4]. This also ensures that our semantics is equivalent to Berry's ones. Finally, we have proved a technique to avoid the *schizophrenia* problem for local variables. A future direction in this area will be to formally define the notions of reactivity and causality [4], so that even the causality analysis can be performed by the theorem prover.

## 4.3. Verifying Quartz Programs

Our major field of interest is the formal verification of properties. We consider temporal logic properties as well as more general higher order logic specifications. The latter are not limited to finite data types, and even allow us to *verify program schemes*. We have already considered some examples, including the Island Traffic Control problem [16], the Mine Pump [24], and the arbitration process of [27].

### 4.3.1 Verification by Theorem Proving

Due to our definitions, Quartz programs can now be used as formulas of the HOL logic. Hence, we can use the whole inference machinery of HOL to verify specifications given in higher order logic. Using theorem proving, we are able to prove invariants, to apply induction on the number of threads or on the data types. In particular, we found it very useful to use the conjunctive form of the control flow as given in lemma 1.

### 4.3.2 Verification by Model Checking

A main application of synchronous programs is to implement complex control behavior with closely interacting threads. For these applications, it is typical that only finite data types are used, so that the entire semantics may be represented by a finite state machine. Therefore, symbolic model checking [7, 9] lends itself well for the verification of these systems. For this purpose, we have combined the HOL temporal logic library [26] with the presented Quartz theory to obtain a model checking tool for verifying temporal properties of Quartz programs. We note that the disjunctive form of the control flow as given in definition 7 is better suited for this purpose than the conjunctive one (cf. lemma 2), since it naturally supports a disjunctive partitioning of the BDDs [8]. For example, we have obtained the following runtimes for the arbitration process of [27] (on a 450 MHz Pentium III with 256 MBytes main memory):

| $n$ | BDD nodes | time [sec.] |
|---|---|---|
| 5 | 10046 | 0.07 |
| 10 | 13486 | 0.28 |
| 15 | 28553 | 0.92 |
| 20 | 49920 | 1.71 |
| 25 | 65837 | 2.96 |

| $n$ | BDD nodes | time [sec.] |
|---|---|---|
| 30 | 93122 | 4.43 |
| 35 | 125201 | 6.32 |
| 40 | 180803 | 8.60 |
| 45 | 197822 | 11.43 |
| 50 | 248658 | 14.75 |

Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE

## 5. Conclusions

We have shown a new way to define the semantics of imperative synchronous programming languages. In particular, we have shown how our Esterel variant Quartz can be easily embedded in a higher-order theorem prover by using this semantics. For this purpose, it is important that our definition of the semantics is based on primitive recursive definitions that can directly be used for embedding Quartz within an interactive theorem prover like HOL. As a result, we can use HOL's inference machinery for reasoning about Quartz, for formal code generation (in particular formal hardware synthesis) [25], and of course, for formal verification of program specifications.

## References


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

[2] M. Baldamus and K. Schneider. Extending Esterel by asynchronous concurrency. In *GI/GMM/ITG Fachtagung zum Entwurf Integrierter Schaltungen*, Darmstadt, Germany, September 1999. VDE Verlag.

[3] 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.

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

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

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[7] C. Berthet, O. Coudert, and J. C. Madre. New ideas on symbolic manipulations of finite state machines. In *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 1990.

[8] J. Burch, E. Clarke, and D. Long. Symbolic model checking with partitioned transition relations. In A. Halaas and P. Denyer, editors, *International Conference on Very Large Scale Integration (VLSI)*, pages 49–58, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[9] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: $10^{20}$ States and Beyond. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–33, Washington, D.C., June 1990. IEEE Computer Society Press.

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

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

[12] K. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Austin, Texas, May 1989.

[13] ECL Homepage. Website, 2000. http://www-cad.eecs.berkeley.edu/

[14] E. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 996–1072, Amsterdam, 1990. Elsevier Science Publishers.

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

[16] K. Fisler and S. D. Johnson. Integrating design and verification environments through a logic supporting hardware diagrams. In *IFIP Conference on Computer Hardware Description Languages and Their Applications (CHDL)*, pages 669–674. IEEE Cat. No. 95TH8102, September 1995. CHDL proceedings pp. 493-696 of the "ACV'95" held August 29 to September 1, 1995, Chiba, Japan.

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

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

[19] D. Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, pages 231–274, 1987.

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

[21] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design – a classification and survey. In M. Srivas and A. Camilleri, editors, *International Conference on Formal Methods in Computer Aided Design (FMCAD)*, volume 1166 of *Lecture Notes in Computer Science*, pages 294–299, Palo Alto, CA, USA, November 1996. Springer Verlag.

[22] T. Melham. Automating recursive type definitions in higher order logic. Technical Report 146, University of Cambridge Computer Laboratory, Cambridge CB2 3QG, England, September 1988.

[23] T. Melham. A package for inductive relation definitions in HOL. In M. Archer, J. Joyce, K. Levitt, and P. Windley, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 350–357, Davis, California, August 1991. IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press.

[24] P. Giridhar, V. Kumar, and M. Joseph. The mine pump control program in Esterel. Computing Sciences Research Technical Report CS-RR-332, University of Warwick, http://www.dcs.warwick.ac.uk, August 1997.

[25] 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.

[26] K. Schneider and D. Hoffmann. A HOL conversion for translating linear time temporal logic to $\omega$-automata. In Y. Bertot, G. Dowek, C. Paulin-Mohring, and L. Théry, editors, *Higher Order Logic Theorem Proving and its Applications*, volume 1690 of *Lecture Notes in Computer Science*, pages 255–272, Nice, France, September 1999. Springer Verlag.

[27] K. Schneider and V. Sabelfeld. Introducing mutual exclusion in Esterel. In *Andrei Ershov Third International Conference Perspectives of Systems Informatics*, Lecture Notes in Computer Science, Novosibirsk, Akademgorodok, Russia, July 1999. Springer Verlag.



Proceedings of the Second International Conference on Application of Concurrency to System Design (ACSD'01)
0-7695-1071-X/01 $10.00 © 2001 IEEE