

Using IP Cores in Synchronous Languages

Jens Brandt, Klaus Schneider and Adrian Willenbücher
Embedded Systems Group
Department of Computer Science
University of Kaiserslautern
<http://es.cs.uni-kl.de>

Abstract

Synchronous programs offer comfortable statements for preemption, which allow statements to abort or suspend other statements. However, while these preemption statements can be used to conveniently describe complex control behaviors in a concise, but yet precise way, they impose very difficult problems for the modular synthesis of systems. For this reason, the integration and re-use of already pre-compiled IP cores has - so far - only been possible in two restricted variants: either by calls to instantaneous procedures of the host language or by calls to asynchronous tasks of the host language. The first variant is restricted to combinational circuits, and the second variant breaks the synchronous design paradigm.

In this paper, we therefore propose the use of wrappers to integrate different kinds of IP cores. These IP cores may not be written in a synchronous language, but can nevertheless be called almost without restrictions in a synchronous program. Our wrappers around the instantiated cores imitate the standard interface of synchronous modules used by compilers, which is the key for our seamless integration. We have implemented the approach in the upcoming version 2.0 of the Averest system and illustrate its use in this paper by a sequential multiplier circuit.

1 Introduction

It is well-known that embedded systems are becoming more and more complex, while their design time is restricted by tight time-to-market constraints. An efficient way to tackle this problem is to re-use already existing components to assemble new systems, which already lead to new design flows like SoC design [6, 21]. This development strategy is mandatory for several reasons: First, reusing components significantly reduces the time to market, and second already proven and tested components generally increase the safety of the overall system. Third, especially in the area of embedded systems, developers need a deep understanding of the application. Typically, only specialized experts have the knowledge and experience for a given system part. These reasons led to the current market structure: all the components of an embedded system are developed by various suppliers, which are then assembled by the distributing company.

In computer science, these components are rarely physical objects. Instead, code bases exist for hardware parts (e.g. processor cores), for software parts (e.g. software libraries) as well as combinations of them (e.g. CAN bus controllers). They are usually distributed as so-called intellectual property (IP) cores. Using these IP cores gives rise to some problems, of which the most challenging one is the support of an adequate infrastructure for their integration¹. In general, there are several alternatives: First, it might be the case that the module is available as

¹See e.g. <http://www.vsia.org> or <http://www.spiritconsortium.org>.

a *soft core* in a specific description language, and the user must use this language in order to use the module. Second, the IP provider may distribute a *hard core* and makes use of a high-level integration mechanism, which allows the integration of several modules written in different languages but using the same integration mechanism. This is based on the principle that there are two levels of a description language: the language in which the module is developed and a language that is merely used for the integration, i.e., a meta-language [9].

Esterel-Studio, an integrated development environment for the synchronous programming language Esterel, follows the second approach² (which we will call shallow embedding in the following) and provides a generation of components conforming to the IP-XACT industry standard [2], which can be subsequently combined with other conforming components from different sources. While this idea is very appealing due to its flexibility, it requires an interaction with the IP cores at a very low level: The usual integration only allows a parallel execution of the IP cores and the developed system, and communication between them is restricted to signals. All other features imperative synchronous languages provide (e.g. sequencing, sophisticated preemption) must be explicitly handled. Therefore, using the module at an arbitrary position in the synchronous program is not possible. Furthermore, all higher data types must be explicitly reduced to simple ones that the integration language supports. For many hardware description languages, this may be more acceptable, since their abstraction level tends to be lower compared to synchronous languages.

In contrast to this shallow embedding, our approach provides a *deep embedding* of IP cores for synchronous languages. Developers can use the synchronous programming language both for the creation of new native modules *and* their integration with proprietary IP cores. This provides the developer with a high-level interface, since all modules can be used without any restrictions in the program (in particular, also in preemption statements). We implemented our integration approach for the imperative synchronous language Quartz by extending the linker of the upcoming version of the Averest system.

This paper is structured as follows: In Section 2, we give a brief overview over the synchronous language Quartz and its main programming paradigms. Section 3 presents our module format, which defines the interface that must be provided by all components to be integrated by the linker (that is used for both hardware and software synthesis). Subsequently, Section 4 defines several classes of IP cores and explains how the required interface can be added to cores of these classes. In Section 5, we illustrate our approach by an example before we draw some conclusions in Section 6.

2 Synchronous Languages

Quartz [14] is an imperative synchronous language derived from the Esterel language [5, 4]. The common paradigm of all synchronous languages is perfect synchrony [10, 1], which means that a sequence of statements can be executed in zero time. Consumption of time must be explicitly programmed by special statements which partition the whole program into macro steps. In the programmer's view, all macro steps take the same amount of logical time. Thus, concurrent threads run in lockstep and automatically synchronise at the end of their macro steps. The introduction of this logical time scale is not only a very convenient programming model, it is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs. Thus, synchronous programs can be executed on ordinary microcontrollers without complex operating systems. As another advantage, the translation of synchronous programs to hardware circuits is straightforward [3, 13]. Furthermore, the concise formal semantics of syn-

²Esterel additionally supports the concept of a 'host language', where functions and procedures written in the host language can be called from the synchronous program. However, host language constructs are not considered by Esterel compilers, and are therefore not verified. Thus, recent efforts try to move as much as possible into the Esterel part and to completely eliminate the host language.

chronous languages makes them particularly attractive for reasoning about program properties. Therefore, synchronous languages are well-suited for the design of safety-critical embedded systems that consist of both, application-specific hardware and software.

In the following, we give a brief overview of the Quartz language core, which is powerful enough to define most other statements as simple syntactic sugar. Due to lack of space, we do not describe the semantics of Quartz in detail, and refer instead to [14] and, in particular, to the Esterel primer [4], which is an excellent introduction to synchronous programming. The Quartz core consists of the following statements, provided that S , S_1 , and S_2 are also core statements, ℓ is a location variable, x is a variable, σ is a Boolean expression, and α is a type:

- `nothing` (empty statement)
- $x = \tau$ and `next(x) = τ` (assignments)
- $\ell : \text{pause}$ (consumption of time)
- `if (σ) S_1 else S_2` (conditional)
- $S_1; S_2$ (sequential composition)
- `do S while(σ)` (iteration)
- $S_1 \parallel S_2$ (synchronous concurrency)
- `[weak] abort S when [immediate](σ)` (abortion)
- `[weak] suspend S when [immediate](σ)` (suspension)
- $\{\alpha x; S\}$ (local variable y with type α)
- $\text{name}(\tau_1, \dots, \tau_n)$ (module instantiation)

There is only one basic statement that defines a control flow location, namely the `pause` statement³. These statements are endowed with unique Boolean valued *location variables* ℓ , which are true iff the control flow is currently at the statement $\ell : \text{pause}$. Since all other statements are executed in zero time, the control flow can only rest at these positions in the program.

There are two variants of assignments, both of which evaluate the right-hand side in the current step. However, immediate assignments $x = \tau$ transfer the value of τ to the left-hand side x immediately, whereas delayed ones `next(x) = τ` transfer the value in the following step. Due to the perfect synchrony paradigm, each variable has exactly one value throughout the whole step, which leads to the well-known causality problems [16, 18, 15, 19]. Another related problem is caused by local variables whose scope is left and reentered within a single macrostep. As only a single value can be assigned to each variable, local variables must be consequently duplicated by any compiler. These problems are also well-known as schizophrenia problems [17, 20].

In addition to the control flow constructs known from other imperative languages (conditional, sequential composition and iteration), Quartz offers synchronous concurrency: $S_1 \parallel S_2$ starts the statements S_1 and S_2 immediately, and both run in lockstep, i. e. they automatically synchronise by a global clock when they hit the following `pause` statement. Furthermore, Quartz offers sophisticated preemption statements: A statement S which is enclosed by an `abort` block is immediately terminated when the given condition σ holds. Similarly, the `suspend` statement freezes the control flow in a statement S . Thereby, two variants of preemption must be distinguished: strong (default) and weak (indicated by the keyword `weak`) preemption. While strong preemption deactivates the data flow of the current step (i. e. the preemption takes place at the beginning of the macro step), weak preemption lets the interrupted statement first execute all its data actions until the end of the current step.

Quartz also supports modular design. An arbitrary synchronous program can be encapsulated into a module, which exposes a set of input and output signals for interaction. In the following, we extend this mechanism to ordinary IP cores. This is not straightforward, since modules can then be instantiated at an arbitrary place in the program. Thus, in addition to parallel execution, a module instantiations can be sequentially chained to other behaviours.

³To be precise, the rarely used immediate forms of `suspend` also have this ability.

Furthermore, it can be located in any abortion or suspension context, which possibly preempts its execution. Hence, it is clear that additional (implicitly defined) control signals are needed, which supervise the execution of these modules. These signals as well as the general structure of compiled modules will be explained in the next section.

3 Averest Intermediate Format

As the starting point for a linker, we use *guarded actions* [7, 8, 11, 12] as a well-established intermediate format, which is used in the compilation process of almost every synchronous language. In particular, guarded actions are also the essential part of the intermediate representation Averest Intermediate Format (AIF) of our Averest system⁴: Guarded actions are the first intermediate result of the compiler, and serve as starting point for hardware and software synthesis [17].

Thus, the modules that we consider for integration are defined by sets of guarded actions. Each one has the form (γ, \mathcal{C}) , where the Boolean condition γ is called the guard and \mathcal{C} is called the action of the guarded action, which are either immediate assignments of the form $x = \tau$ or delayed assignments of the form $\text{next}(x) = \tau$.⁵

The semantics of these guarded actions is defined as follows: In each macro-step, the guards of all actions (of all variables) are checked simultaneously. If a guard is true, the right-hand side of the action is immediately evaluated. Immediate actions assign the computed value immediately to the variable on the left-hand side of the assignment, while the updates of delayed actions are deferred to the following macro-step. If no action is active in a reaction, event variables are reset to their default values, while memorised (registered) variables maintain their previous values.

In principle, linking of several modules on this level only consists of collecting their guarded actions and connecting the variables which are exposed at the *data interface*. However, module instantiations may occur at arbitrary points in the synchronous program so that further signals are needed to control the execution of the individual modules. Hence, in addition to the data variables exposed by the interface at source-code level, each AIF module M implicitly provides the following *control interface* inputs:

- $\text{go}^{\text{surf}}(M)$ activates only the combinational part of the module.
- $\text{go}^{\text{depth}}(M)$ activates and enters the module ($\text{go}^{\text{depth}}(M) \rightarrow \text{go}^{\text{surf}}(M)$ always holds).
- $\text{abrt}(M)$ aborts the control flow.
- $\text{susp}(M)$ suspends the control flow and has higher priority than $\text{abrt}(M)$.
- $\text{prmt}(M)$ kills the data flow (usually a result of a surrounding abort- or suspend-statement), and all actions of M are deactivated.⁶

In exchange, the following signals are output by each module M to give the surrounding module information about its own execution status:

- $\text{inst}(M)$ is true iff the module is combinational (instantaneous).
- $\text{insd}(M)$ indicates whether a control flow in the module is active, i. e. whether the module has been started in a previous cycle ($\text{go}^{\text{depth}}(M) = \text{true}$) and its computations have not yet terminated.
- $\text{term}(M)$ is true iff the module completes its computations and terminates voluntarily in this macro step.

Due to the lack of space, the motivation for the definition of this specific set of signals cannot be given at this point. Instead, we refer to other publications [17, 14] which present the compilation of synchronous programs in more detail and thereby reveal the motivation.

⁴See <http://www.averest.org>.

⁵In principle, each IP core can be decompiled into guarded actions so that it can be used like any other module, e. g. for simulation and analysis.

⁶The signals $\text{abrt}(M)$, $\text{susp}(M)$ or $\text{prmt}(M)$ only affect the non-combinational part of the module. If a module M should be aborted or suspended in its initial step, $\text{go}^{\text{depth}}(M)$ and possibly $\text{go}^{\text{surf}}(M)$ are set to false.

4 Integrating IP Modules

In principle, an IP core must provide the full data interface and control interface (as defined in the previous section) in order to be deeply integrated in a synchronous program. As this is not generally the case, a wrapper must be added, which translates the interface required by the synchronous language environment to the specific one of the IP core. Then, the resulting combination of wrapper and core (which will be called *IP module* in the following) cannot be distinguished from modules written in a synchronous language.

This wrapper has to fulfill the following basic requirements: First, it must actually instantiate the core and control it according to its input control signals. Second, it must provide the logic for the output control signals and data signals, and third, it must be written in a language which is understood by the synthesis tool (typically VHDL or Verilog). The following paragraphs outline these requirements.

4.1 Control Interface

In order to maintain the advantages of synchronous languages for verification, wrappers for IP cores should respect the well-formed properties of synchronous modules. In particular, modules are expected to be cooperating, i. e. such a module will terminate if it is aborted from the outside context, and will deactivate its data flow if the preemption context demands this deactivation. If these properties have been checked for an IP core, verified properties referring to the native part of the system can still be guaranteed for the entire system. A very simple way to check the well-formedness for a developer is to ensure that he would be able to give a synchronous program that models the same behaviour as the IP core.

Integrated cores usually provide some control signals that allow the wrapper to influence their execution. In the addition to the obligatory clock signal, many circuits expose a *reset* (*rst*) and a *clock enable* (*ce*) signal. Quartz modules usually do not need them: starting a module brings it to a predefined state, and a suspend-statement can be placed around it to temporarily stop its execution. Therefore, designers should hide these signals inside the wrapper as will be shown in Section 4.4.

Nevertheless, we are sure that this may not always be possible or advantageous. For example, some complex cores provide reset signals for different parts of their functionality; these should be made explicit inputs of the module. Others do not accept inputs in the same cycle in which they are reset; this means that the corresponding modules do not accept inputs in their first step. Designers will have to make sensible choices, depending on the core itself, as well as on the desired level of abstraction.

4.2 Data Interface

In contrast to the control signals, the logic for the translation of the data interface, which contains the actual inputs and outputs of the core, is straightforward. The wrapper usually does not have to look at the inputs; however, depending on the class of the core, it has to look at some or all of the outputs in order to determine whether the core has finished its task and the module terminates.

The only thing the wrapper needs to implement is the selection of variable values. Since variables can be shared between several modules, and since each one potentially assigns new values to it, write operations must be arbitrated. The synchronous semantics requires that there are no conflicting assignments in a single step. Hence, the problem only consists of selecting the module that determines the value of the variable in the current step. To this end, the wrappers are given in addition to the input variables x_1, \dots, x_m the following signals for each output variable $y_i \in \{y_1, \dots, y_n\}$

- y_i^{in} is value of the output variable y_i determined by the surrounding module. The wrapper has to forward this value to the core if it does not determine the value itself, since it may read the value.
- y_i^{core} is the value of the output variable y_i determined by the core.
- y_i^{out} is the actual value of the output variable y_i .

In the following two subsections, we show how to build a wrapper that provides these control and data interfaces. Thereby, we distinguish two fundamental classes: combinational or sequential cores. Thus, we strictly divide them according to this characteristic. In fact, the module description could even include a flag indicating the class, thereby potentially allowing the synthesis tool to perform certain optimizations.

4.3 Combinational Modules

Combinational modules do not have a clock signal. They are instantaneous and can be modeled in synchronous languages by using immediate actions only. Since the module does not have a clock signal, the control flow is never inside of them. Hence, the control signals $\text{go}^{\text{depth}}(M)$, $\text{abrt}(M)$, $\text{susp}(M)$ and $\text{prmt}(M)$ do not have any influence on the behaviour, the wrapper can generate the following output signals independently of the concrete IP core:

- $\text{inst}(M) = \text{true}$
- $\text{insd}(M) = \text{false}$
- $\text{term}(M) = \text{false}$

All data outputs of the core are conditionally forwarded. A multiplexer selects between the value generated by the core and the value generated somewhere outside the core, where the activation condition of the module $\text{go}^{\text{surf}}(M)$ is responsible for the selection.

- $y_i = (\text{go}^{\text{surf}}(M) \Rightarrow y_i^{\text{core}} | y_i^{\text{in}})$

4.4 Sequential Modules

The integration of sequential modules is more complicated. Before they can be used, some cores have to be reset, while others either do not need a reset or can be reset in the same cycle as they accept their first input. In all these cases, the wrapper hides the reset signal from the user; the representing Quartz module is assumed to start with a sequence of empty macro steps. Note that the user still has to consider the timing in the interaction.

In order to compute the control signal outputs, the wrapper for sequential modules needs an additional register l which is true whenever the control flow is inside of it. The register is set when the control flow enters the module ($\text{go}^{\text{depth}}(M) = \text{true}$) or moves within the module. It is cleared if the module terminates or a preemption takes place without re-entering the module at the same time.

This information is needed for the $\text{insd}(M)$ control signal itself, as well as for the $\text{term}(M)$ control signal. For the special case of a core that never terminates, the latter is constantly false. For all other cases, the termination of the core must be determined from its output signal. In general, most cores which terminate at some point provide a *ready* signal which notifies the wrapper of this condition; otherwise the wrapper has to deduce this condition in another way (e. g. by counting the number of cycles since the core was started).

- $\text{inst}(M) = \text{false}$
- $\text{insd}(M) = l$.
- $\text{term}(M) = l \wedge \text{rdy}$ if the core terminates at some point, otherwise $\text{term}(M) = \text{false}$.
- $\text{next}(l) = \text{go}^{\text{depth}}(M) \vee (\text{insd}(M) \wedge \neg(\text{abrt}(M) \vee \text{term}(M))) \vee (\text{insd}(M) \wedge \text{susp}(M))$.

The module is active during the next macro step if the module is started (again) or if it is already active and it neither terminates nor it is aborted. The last part of the condition covers the case when the module is suspended and active: then it will always be active in the next macro step, too.



Figure 1: Module Duplication Examples

- $y_i = ((\neg \text{prmt}(M) \wedge (\text{insd}(M) \vee \text{go}^{\text{surf}}(M))) \Rightarrow y_i^{\text{core}} | y_i^{\text{in}})$. The outputs of the core define the values of the outputs of the module in the current macro step iff the data flow is not killed, and the module is active. This condition can optionally be combined with other predicates (for example a data-valid signal from the core) or with different conditions for different variables, in order to control more precisely in which macro steps the module defines the values of the output variables.

The following signals are used by the IP core itself:

- $\text{ce} = \neg \text{susp}(M)$. If the module is suspended, its state is frozen by clearing the clock enable signal of the core. However, its outputs may still react to changes at its inputs (if $\text{prmt}(M) = \text{false}$). Note that cores without a clock enable signal cannot be instantiated inside a suspend statement, since it generally provides the only possibility to suspend the execution of a previously started core.
- $\text{rst} = \text{go}^{\text{depth}}(M)$. If a reset signal exists, it is set whenever the module is entered.

4.5 Weak Suspend

A challenging operation for the wrapper is the weak suspend, which halts the control flow and leaves the data flow unaffected. Many cores only have one clock-enable signal for the both parts. This is mainly due to the lack of such a distinction of them in netlists and most hardware description languages (like VHDL or Verilog). Consequently, these languages cannot distinguish between strong or weak preemption either.

From the practical side, it is difficult to imagine a correct application of weak suspend to a core. Therefore, we decided to freeze the state, but allow changes in the inputs to affect the outputs (i. e. disable delayed-assignments, but enable immediate-assignments). This is a sensible choice, although the corresponding Quartz code for the module would look very bizarre (the state would be modelled as control flow). However, this is not a conceptual problem, since we must only ensure that such a model exists.

4.6 Duplication of Modules

Duplication of modules arises from our support to embed IP cores at arbitrary parts of the program. This is necessary if a module is immediately restarted in the step when it terminates. This can happen if the module is instantiated in a loop, where the last macro step of module M overlaps with the first one of next iteration. A similar but more general situation is depicted in Figure 1: here, due to the weak abort statement embracing the module instantiation, any macro step (including the first one) of the module M can be potentially active at the same time as the first step of the next loop iteration.

This situation (which is known as schizophrenia in synchronous languages [17, 20]) can be avoided by the programmer by just starting the loop body with a `pause` statement. Nevertheless, since we aim at not restricting the expressiveness of synchronous languages, we do not refuse to compilation of these programs and allow the instantiations of IP cores at the beginning of loops, which makes the duplication of the combinational part necessary in general.

For native Quartz modules, this duplication is automatically done by the compiler. For IP cores, however, the combinational part cannot be extracted in general so that a duplication of the whole IP core is required. In order to do this expensive duplication only when absolutely needed, we add two attributes to the module description. The first one, *dupEnd*, means that a core cannot be started ($\text{go}^{\text{surf}}(M)$ is true) in the same step as it terminates ($\text{term}(M)$ is true). This attribute enforces duplications in situations like the first example. The second one, *dupAny*, indicates combinational outputs. If this is the case, a core needs to be duplicated in situations like the second example.

5 Example

To illustrate the design of a wrapper for sequential modules, this section contains the corresponding code in VHDL for a multiplier. This multiplier is not pipelined and has a latency of multiple cycles; its inputs as well as its outputs are registered. Figure 2 lists the ports provided by the core.

The code for the wrapper is given in Figure 3. The control signals are generated exactly in the same way as described in the previous sections. All data inputs of the core are registered, hence there will be no conflict if the module terminates (or is aborted) and is restarted in the same macro step. Hence, both *dupEnd* and *dupAny* are false. The resulting module can then be used as follows:

```

sum = 0;
i = 0;
while(i < n) {
  MultiplierModule(A[i], B[i], P);
  next(sum) = sum + P;
  i = i + 1;
}

```

The programmer only has to handle the inputs and outputs; the compiler and the wrapper handle the reset of the IP core, managing the control signals and dealing with (possibly variable) latencies. Since *dupEnd* of the module is false, *MultiplierModule* (and the multiplier itself) has to be instantiated only once, although the last macro step in one loop iteration is active at the same time as the first one in the next iteration.

6 Conclusions

In this paper, we proposed a method to embed IP cores into synchronous (Quartz) programs. This core can be given in any form suitable to the synthesis tool which produces the netlist of

Signal	Direction	Description
<i>clk</i>	in	clock signal
<i>ce</i>	in	clock enable signal
<i>valid</i>	in	true iff new input data is available; stops an older computation if necessary
<i>A</i>	in	first factor
<i>B</i>	in	second factor
<i>Y</i>	out	result (product)
<i>rdy</i>	out	true iff the result is available

Figure 2: HDL Interface of the Example Multiplier

```

entity MultiplierModule is
  port (
    clk: in std_logic ;
    rst : in std_logic ;

    -- control signals
    go_surf, go_depth: in std_logic ;
    kill, susp: in std_logic ;
    preempt: in std_logic ;

    insd: out std_logic ;
    inst : out std_logic ;
    term: out std_logic ;

    default_Y: in std_logic_vector ( 31 downto 0 );

    -- data signals ( visible in Quartz code):
    A, B: in std_logic_vector ( 31 downto 0 );
    Y: out std_logic_vector ( 31 downto 0 )
  );
end MultiplierModule;

architecture synt of MultiplierModule is
  signal l_1: std_logic ;
  signal define_vars : std_logic ;

  signal core_y: std_logic_vector ( 31 downto 0 );

  signal rdy, ce, valid: std_logic ;
  signal inst_i, insd_i, term_i: std_logic ;

component Multiplier is
  port (
    signal clk, ce: in std_logic ;
    signal valid: in std_logic ;
    signal A, B: in std_logic_vector ( 31 downto 0 );
    signal Y: out std_logic_vector ( 31 downto 0 );
    signal rdy: out std_logic
  );
end component;

begin
  -- wrapper:
  ce <= susp;
  valid <= go_depth;

  insd <= insd_i ;
  inst <= inst_i ;
  term <= term_i;

  insd_i <= l_1 ;
  inst_i <= '0';
  term_i <= insd_i and rdy;

  define_vars <= not preempt and (insd_i or go_surf);
  Y <= core_Y when define_vars = '1' else default_Y;

  process( clk )
  begin
    if rising_edge( clk ) then
      if rst = '1' then
        l_1 <= '0';
      else
        l_1 <= go_depth or (insd_i and
          not ( kill or term_i)) or (insd_i and susp);
      end if;
    end if;
  end process;

  -- instantiation of the core:
  core: Multiplier
  port map (
    clk => clk, ce => ce,
    valid => valid,
    A => A, B => B,
    Y => core_Y,
    rdy => rdy
  );
end synt;

```

Figure 3: Complete Code of the Multiplier Example

the design; typically this includes netlists, Verilog and/or VHDL. We achieved this by defining a wrapper around the instantiation of the core. This wrapper in turn is part of a module description, together with two attributes that give hints to the Quartz compiler regarding the efficient use of chip resources.

Although the wrapper has to be written for every IP core, this task requires little work and is straightforward in practice. Furthermore, the wrapper hides certain details like control signals, duplication and in some cases even latency issues.

We expect that the ability to deeply embed IP cores, especially those for which no or no efficient equivalent representation exists, will extensively contribute to the utilisability of synchronous languages. The effort to implement this is low for synchronous language compilers which already support modular compilation, since both approaches require the same interface and are handled very similarly by the calling module.

References

- [1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
- [2] V. Berman. Standards: The P1685 IP-XACT IP metadata standard. *IEEE Design and Test of Computers*, 23(4):316–317, 2006.
- [3] G. Berry. A hardware implementation of pure Esterel. In *Workshop on Formal Methods in VLSI Design*, Miami, Florida, 1991.
- [4] G. Berry. The Esterel v5 language primer. <http://www-sop.inria.fr/esterel.org/>, July 2000.
- [5] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
- [6] B. Bouyssounouse and J. Sifakis. *Embedded Systems Design*, volume 3436 of *LNCS*. Springer, 2005.
- [7] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison Wesley, Austin, Texas, May 1989.
- [8] D.L. Dill. The Murphi verification system. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification (CAV)*, volume 1102 of *LNCS*, pages 390–393, New Brunswick, USA, 1996. Springer.
- [9] A. Habibi and S. Tahar. A survey on system-on-a-chip design languages. In *IEEE 3rd International Workshop on System-on-Chip (IWSOC'03)*, Calgary, Canada, 2003. IEEE Computer Society.
- [10] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.
- [11] H. Järvinen and R. Kurki-Suonio. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.
- [12] L. Lamport. The temporal logic of actions. Technical Report 79, DEC, 1991.
- [13] F. Rocheteau and N. Halbwachs. Implementing reactive programs on circuits: A hardware implementation of LUSTRE. In *Proceedings of the Real-Time: Theory in Practice*, volume 600 of *LNCS*, pages 195–208. Springer, 1991.
- [14] K. Schneider. The synchronous programming language Quartz. Internal report, Department of Computer Science, University of Kaiserslautern, 2008.
- [15] K. Schneider and J. Brandt. Performing causality analysis by bounded model checking. In *Conference on Application of Concurrency to System Design (ACSD)*, Xi'an, China, 2008. IEEE Computer Society.
- [16] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pages 179–189, Washington, DC, USA, 2004. ACM.
- [17] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 153(4):71–97, 2006.
- [18] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, 2005. IEEE Computer Society.
- [19] T.R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, Berkeley, CA, USA, 1996.
- [20] O. Tardieu and R. de Simone. Curing schizophrenia by program rewriting in Esterel. In *International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 39–48, San Diego, California, USA, 2004. IEEE Computer Society.
- [21] Richard Zurawski, editor. *Embedded Systems Handbook*. CRC Press, 2005.