From Synchronous Guarded Actions to SystemC

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

Abstract
In this paper, we present an automatic translation from synchronous guarded actions to SystemC. Synchronous guarded actions are often used as intermediate code representation of synchronous languages like Esterel and Quartz. The obtained SystemC modules can be seamlessly integrated into existing simulation environments, e.g. for fast simulation or the generation of virtual hardware prototypes. Since synchronous guarded actions may contain causality cycles (if actions have an immediate effect on their guards), we must generate a dynamic schedule for the SystemC modules in order to preserve the original semantics of the guarded actions. We implemented the translation procedure within our Averest framework and validated it by several benchmarks that contain difficult causality cycles or reincarnations of local variables.

1 Introduction
System description languages like SystemC [13, 16] and synchronous languages [3, 10] like Esterel [4] are already in use for the efficient development of modern hardware-software systems. The common goal of these languages is to establish a model-based design flow, where different design tasks like simulation, verification and synthesis (of both hardware and software) can be performed on the basis of a single system description.

However, these languages have different underlying models of computation [9, 14, 15]: The execution of a program written in a synchronous language like Esterel is divided into macro steps that correspond with single reactions that are triggered by a common clock similar to a synchronous hardware circuit. Each macro step is divided into finitely many micro steps that are all executed in zero time and within the same variable environment (micro steps correspond to signal flows during a cycle of a synchronous circuit). Hence, the execution of a synchronous program is driven in a cycle-based manner. Due to the instantaneous reaction of micro steps, causality problems may occur if actions modify variables whose values are responsible for triggering that action. In order to analyze the causality of programs, a fixpoint iteration has to be performed to compute the correct reaction of a macro step. It is well-known that this fixpoint iteration corresponds to ternary simulation [7] of the corresponding hardware circuits. Moreover, it has to be remarked that Esterel compilers usually perform this fixpoint analysis at compile time, so that (1) runtime-efficient code is generated and (2) it is known at compile time that the iteration finally terminates with known values.

In contrast, SystemC follows the discrete-event semantics [8] that are well-known from hardware description languages like VHDL [11] and Verilog [12]: A SystemC program consists of a set of sequential processes that are either methods, asynchronous processes or synchronous processes. Methods are special cases of asynchronous processes that do not have wait statements. Asynchronous processes are triggered by events, i.e. they are resumed whenever variables on which the process depends are modified, and they are repeatedly resumed as long as such variable changes are observed.
For this reason, the execution of the asynchronous processes also determines a fixpoint computation that terminates as soon as a fixpoint is found. After this, the synchronous processes are executed once to complete the simulation cycle.

In a previous paper [6], we already outlined the differences and similarities of SystemC and synchronous languages. In particular, we defined classes of systems that can be easily described in both models of computation in a way that allows one to structurally translate these descriptions into each other. This paper has a more general scope: it provides a complete translation of synchronous programs to SystemC so that modules written in synchronous languages like Esterel can be smoothly integrated with SystemC. We implemented this translation for our synchronous language Quartz [20] and applied it successfully to some standard benchmarks [5, 20, 24, 25] to reason about programs with difficult causality cycles and schizophrenia problems.

The benefits of our translation are manyfold. First, it gives a detailed insight in the relationship of both models of computation so that the research done in these communities can be combined, and research/engineering results can be shared. For example, solutions that have been developed to solve problems for synchronous languages (like causality problems) can be transferred to solve corresponding problems of SystemC. In addition, these problems can now also be analyzed by means of fast simulation using the SystemC simulation. Conversely, certain runtime problems of SystemC programs can be solved by means of a fixpoint analysis that has been originally developed for synchronous languages.

Of course, verification results obtained for a synchronous program are still valid after its translation to SystemC, since our translation preserves the semantics. From a practical perspective, a major advantage of our translation is the integration of synchronous components in a SystemC description while preserving the synchronous semantics. For example, this can be used for fast simulation or the generation of virtual prototypes. In addition to a very efficient simulation, the rich library of SystemC components including many microprocessors becomes available for synchronous programming.

Integration of different models of computation has gained a lot of interest in recent years. The pioneering work of Lee et. al. [1, 9, 15] introduced the tagged signal model to capture different models of computation like synchronous languages, data-flow process networks and discrete-event languages. An entire textbook of Jantsch as well as many of his papers consider this topic [14]. Furthermore, a new research branch in this domain has been entered by the construction of GALS (globally asynchronous, locally synchronous) components. To this end, this research defined (weak) endochrony and isochrony as properties allowing to embed synchronous modules in asynchronous environments [2, 17, 18]. In this spirit, our translation is not only a translation between languages, rather it is a translation between models of computation.

It has to be noted that there are translations from synchronous languages to sequential C programs, and hence, also to SystemC. However, translating to a sequential program eliminates the originally available concurrency in the program which is, in contrast, even increased by the translation presented in this paper.

The rest of the paper is organized as follows: In Section 2, we give an introduction to perfectly synchronous systems and their semantics. Section 3 presents our translation and illustrates it with the help of a small example. Section 4 discusses the causality problem of perfectly synchronous systems and demonstrates the corresponding effects on SystemC programs. Finally, the paper ends with a short summary in Section 5.

2 Synchronous Guarded Actions

Synchronous systems [3, 10] divide their computation into single reactions. Within each reaction, new inputs are read from all input ports, and new outputs are generated on all output ports with respect to the current state of the system and the inputs. Furthermore, the reaction determines the state for the next reaction. It is very important for synchronous languages that variables do not change during the
\[
\begin{align*}
  a &\Rightarrow y = x + 1 \\
  \neg a &\Rightarrow x = y + 2 \\
  \text{true} &\Rightarrow z = x + y \\
  a &\Rightarrow x = \neg y \\
  a &\Rightarrow y = x \\
  a &\Rightarrow x = \text{true} \\
  a &\Rightarrow x = \text{false}
\end{align*}
\]

Figure 1: Synchronous Guarded Actions

**macro step.** For this reason, all micro steps are viewed to be executed at the same point of time (as they are executed in the same variable environment).

In general, there are two kinds of synchronous systems (similar to Mealy and Moore machines): The so-called *clocked synchronous systems* make their outputs available only in the following macro step. System description languages implement this restriction by either forbidding to read module outputs or by deferring the actual update of output variables to the next macro step. In contrast, *perfectly synchronous systems* make their outputs immediately available, so that they can already be read in the course of the macro step that produces them.

For instance, the imperative synchronous languages Esterel [4] and Quartz [19, 20, 23] follow the perfectly synchronous paradigm. Due to the instantaneous update of values, programs written in these languages may suffer from causality cycles [5, 24, 25]. Compilers check the causality of a program at compile time with a fixpoint analysis that essentially corresponds to those used for checking the speed-independence of asynchronous circuits via ternary simulation [7]. This fixpoint iteration represents exactly the way a synchronous hardware circuit works: all computations and communications within a clock cycle happen more or less simultaneously. In practice, this means that the execution follows the data dependencies between the actions.

In the following, we abstract from concrete synchronous programming languages, which differ in the provided statements and description styles. As a starting point, we rather use *synchronous guarded actions* which are the essential part of the intermediate representation AIF (Averest Intermediate Format) of our Averest system. In our Averest system, synchronous guarded actions are the result of the first compilation phase, and are the starting point for hardware and software synthesis.

Thus, the systems we consider in the following are defined by sets of guarded actions. Each guarded action has the form \(\langle \gamma \Rightarrow C \rangle\), where the Boolean condition \(\gamma\) is called its guard (or the trigger) and \(C\) its action. In perfectly synchronous systems, actions are either immediate assignments of the form \(x = \tau\) or delayed assignments of the form \(\text{next}(x) = \tau\). In Esterel, this choice is made by the declaration of the output variable, but in Quartz, it is possible to perform both immediate and delayed actions on the same variable. After the compilation of a program, we have for each writeable variable \(x\) of the system a set of immediate and delayed actions:

\[
\langle \gamma_1 \Rightarrow x = \tau_1 \rangle, \quad \ldots, \quad \langle \gamma_p \Rightarrow x = \tau_p \rangle, \\
\langle \chi_1 \Rightarrow \text{next}(x) = \pi_1 \rangle, \quad \ldots, \quad \langle \chi_q \Rightarrow \text{next}(x) = \pi_q \rangle
\]

The semantics of synchronous 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 of 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, a variable maintains its previous value.

Synchronous guarded actions are always deterministic, because there is no choice among activated guarded actions. Instead, all of the enabled actions must be fired. Hence, any system is guaranteed to produce the same outputs for the same inputs. However, forcing conflicting actions to fire simultaneously leads to problems. To illustrate the effect of immediate feedback, consider the examples in Figure 1. The guarded actions in the first column have the following behavior: Depending on \(a\), one of the first two actions is executed, which is immediately seen by the last action. Hence, if \(x\) has been 1 in the preceding step and \(a\) is currently true, the actions compute \(y = 2\) and \(z = 3\) in the same macro step. Problematic cases are shown in the second and in the third column of Figure 1. They represent so-called causally incorrect programs, which have either no or a nondeterministic behavior. This is a
well-studied problem for synchronous systems and many analysis procedures have been developed to spot and eliminate these problems [21, 22, 25, 26]. In the following section, we assume that a program is causally correct and that for each variable at most one action is active in a macro step. We will have a closer look at semantic challenges in Section 4.

3 Translating Synchronous Guarded Actions to SystemC

The simulation semantics of SystemC is based on the discrete-event model of computation [8], where reactions of the system are triggered by events, i.e., changes of values of the variables. All threads that are sensitive to a specified set of events are activated and produce new events during their execution. Updates of variables are not immediately visible, but become synchronously visible in the next delta cycle. The structural translations of synchronous languages to SystemC (like the one presented in [6]) were based on this observation and were therefore limited to clocked synchronous systems.

In this paper, we follow a completely different approach, which is based on the guarded actions as described in the previous section. We start the translation by the definition of a global clock that drives the overall computation of the synchronous program. In SystemC, this clock is implemented by a single \texttt{sc\_clock} at the uppermost level, and all other components are connected to this clock. Hence, the translations of the macro steps of the synchronous program in SystemC are triggered by this clock, while the micro steps are triggered by signal changes in the delta cycles. For this reason, input and output variables of the synchronous program are mapped to input signals (\texttt{sc\_in}) and output signals (\texttt{sc\_out}) of SystemC of the corresponding type.

3.1 Preparing Guarded Actions

The translation of the synchronous guarded actions to SystemC processes is however not as simple as one might expect. In general, we have to map the guarded actions to SystemC methods or asynchronous threads so that the simulation core is able to create a dynamic schedule according to their data dependencies. The basic idea is therefore to map guarded actions to methods which are sensitive to the read variables so that the guarded action is re-evaluated each time one of the variables it depends on changes.

However, there are some problems due to the differences between SystemC and the synchronous semantics, which require that we have to process the guarded actions before the actual translation to SystemC. A first problem is that SystemC does not allow a signal to have multiple drivers. Thus, all actions writing a signal must be implemented in the same thread, and we must group all actions of the system by their target variables. We solve this problem by the introduction of a so-called carrier variable \texttt{x′} for every variable \texttt{x}. The immediate assignments implemented in SystemC directly work on \texttt{x} as in the synchronous program, but the delayed assignments work on the carrier variable \texttt{x′} in SystemC. A new default action is added for the original variable \texttt{x} so that values are transferred from \texttt{x′} to \texttt{x} if no immediate assignment takes place.

Second, the SystemC simulation semantics can lead to spurious updates of variables, since threads are always triggered if some variables in the sensitivity list have been updated. Since the values of such a run are not necessarily the final values for the current simulation time (reached in later delta cycles), actions may be spuriously activated. In these cases, the translated system explicitly needs to correct the update by an additional action. This action reflects the so-called reaction to absence, which usually just maintains the old value. It must be translated to an explicit action that assigns the previous value in the SystemC implementation if no other action writes to the variable of the corresponding SystemC thread.

Both problems can be handled by the following transformation that ensures that for each variable exactly one action fires in each step. The resulting code is commonly referred to as equational code [20],
Figure 2: Preparing Synchronous Guarded Actions for SystemC Synthesis

Boolean equations or simply hardware circuits in the context of synchronous languages. Figure 2 sketches the structure for each writable variable \( x \) of the system.

The construction of actions that fulfill the requirements presented above is more difficult than expected, since a straightforward construction would refer to values of the trigger conditions of immediate assignments at the next point of time, i.e. in the future. As already mentioned above, we therefore introduce an auxiliary variable \( x' \) called the carrier of \( x \) to capture delayed assignments at the previous point of time. Before considering the equations for \( x' \), let us consider the invariant for \( x \) in terms of \( x' \): Clearly, we have to demand that \( x \) equals to \( \tau_i \) whenever the guard \( \gamma_i \) of an immediate assignment \( x = \tau_i \) holds. In case no guard of an immediate assignment to \( x \) holds, we have to distinguish whether \( x \) has been assigned a value in the previous step or not, which is both covered by the equations for the carrier variable \( x' \).

The meaning of \( x' \) is as follows: \( x' \) captures all of the delayed assignments \( \text{next}(x) = \pi_j \) to \( x \), that is whenever \( \text{next}(x) = \pi_j \) is executed, we evaluate the right hand side \( \pi_j \) at the current point of time and assign this value to \( x' \) (not yet to \( x \)) at the next point of time. Hence, \( x' \) is determined by the delayed assignments to \( x \). This leaves open what the initial value of \( x' \) should be, so we additionally define the initial value of \( x' \) is the default value of \( x \).

By this definition of the initial value of \( x' \), the initial value of \( x \) is correct. In later macro steps, if one of the immediate assignments to \( x \) is enabled, then this assignment determines the value of \( x \) at this point of time as given by the invariant equation for \( x \). Otherwise, a delayed assignment \( \text{next}(x) = \pi_j \) may have been executed at the previous point of time. If so, then \( x' \) has now the value that has been obtained by evaluating \( \pi_j \) at the previous point of time, and the invariant equation takes this value via \( x' \).

Finally, if neither an immediate assignment \( x = \tau_i \) is currently executed nor a delayed assignment \( \text{next}(x) = \pi_j \) has been executed at the previous point of time, we have to refer to the reaction-to-absence. Note that the generation of equational code assumes that the program does neither suffer from causality problems nor from multiple writes to the same variable in a macro step. As already mentioned, these problems can be solved by a causality analysis on the guarded actions.

Our translation also makes use of some optimizations that can be applied also in other code generation schemes:

- **initialization, then only delayed actions**: The simplest case are variables that are initialized in the first step and subsequently only modified by delayed actions. This is a frequent case, e.g. all control-flow variables of synchronous programs are implemented in this way. For these variables, we do not need an additional carrier variable and all modifications can be subsumed in a single clocked thread.

- **only immediate actions**: There is no need to generate carrier variables for event variables (in the sense of synchronous programs), since these variables are memoryless. They are determined in each step according to the current inputs and outputs and therefore, they are essentially abbreviations of program expressions.
void Module::compute_x () {
    while (true) {
        if (γ1)
            x.write (τ1);
        else if (γ2)
            x.write (τ2);
        ...
        else
            x.write (τn);
        wait () ;
    }
}

void Module::compute_delayed () {
    /* delayed actions for x */
    if (γ1)
        x.write (τ1);
    else if (γ2)
        x.write (τ2);
    ...
    else
        x.write (τn);
    /* delayed actions for y */
    ...
    wait () ;
}

Figure 3: Translation of Immediate and Delayed Actions

3.2 Translating Guarded Actions

After the transformation described in the previous subsection, the further translation of the imme-
diate guarded actions \(\langle \gamma \Rightarrow x = \tau_i \rangle\) is straightforward: We translate each group of actions into an
asynchronous thread in SystemC, which is sensitive to all signals read by these actions (variables
appearing in the guards \(\gamma_i\) or in the right-hand sides \(\tau_i\)). Thereby, all actions are implemented by an
if-block except for the last one, since it is ensured by the previous transformation that exactly one
actions fires. Since the immediate actions should become immediately visible, the new value can be
immediately written to the variable with the help of a call to \(x.write(\ldots)\). Analogously, the
evaluations of the guard \([\gamma_i]\) and the right-hand side of the assignment \([\tau_i]\) make use of the read
methods of the other signals. The left-hand side of Figure 3 shows the general structure of such a
thread and shows a small example.

Delayed actions \(\langle \gamma \Rightarrow next(x) = \pi_j \rangle\) must be handled differently: While the right-hand side is
immediately evaluated, the assignment should only be visible in the following macro step and not yet
in the current one. Hence, they do not take part in the fixpoint iteration. They are evaluated at the end
of the iteration, when all variable values for the current step are known. We implement this behavior
in a single clocked thread, because there is no need to split or to schedule these actions. This also
ensures that signals changed by the delayed actions do not affect the current fixpoint iteration, but they
can affect the fixpoint iteration of the next step, which starts after the clock signal. The right-hand
side of Figure 3 shows the general structure and a small example for that part of the translation.

3.3 Example

An example is given by the guarded actions shown in Figure 4 (a). The system described by these
actions takes the two inputs \(i\) and \(sel\) and produces three outputs \(l_1\), \(l_2\) and \(l_3\). The outputs only change
their values in the steps where \(sel = true\) holds. Then, the outputs provide the last three values of the
input \(i\). So the outputs provide a snapshot of the input trace. In Figure 4 (b) the system transformed
to equational code is shown. The system is extended by three local variables carrying the values of
the outputs to the next step.

The translation of the guarded actions to SystemC together with some example code is shown in
Figure 5. The driver needs to provide the inputs before the clocked thread, which updates the signals
for the next step, runs. Otherwise, the outputs are computed with wrong signal values. A sample
output trace of the example is shown in Figure 6. The boxes indicate the trace snapshots provided at
the outputs of the module when the inputs \(sel\) is true.
true ⇒ next\((a_1) = i\)  
true ⇒ next\((a_2) = a_1\)  
true ⇒ next\((l'_1) = l_1\)

(a) Original Guarded Actions

true ⇒ next\((a_1) = i\)  
true ⇒ next\((a_2) = a_1\)  
true ⇒ next\((l'_2) = l_2\)

(b) Transformed Guarded Actions

true ⇒ next\((a_1) = i\)  
true ⇒ next\((a_2) = a_1\)  
true ⇒ next\((l'_3) = l_3\)

4 Semantic Challenges

As mentioned in Section 2, the compilation of perfectly synchronous programs is a difficult problem, since it requires to resolve the causality cycles at compile time in order to guarantee a deterministic code generation.

A first problem is thereby given by write conflicts, which occur if different values are assigned to the same output variable in the same macro step. This is the case if the guards of several immediate or several delayed actions of an output variable become true in the same step, but it may also be the case that an immediate action is in conflict with a delayed action of the preceding macro step. As long as the actions assign the same value, we do not regard multiple writes as a conflict. However, if the values are different, it is impossible to derive a consistent behavior. Hence, programs with write conflicts have to be discarded by a compiler, since write conflicts would be hidden in our SystemC translation by implicit priorities of actions. Fortunately, write conflicts are detected during the causality analysis of the compilers.

Another problem is posed by causality cycles. The guards of an action may directly or indirectly depend on the variable assigned by its action. These causality cycles are also a well-known problem for synchronous programs, and algorithms have been developed by various researchers that can be used to analyze the situation and to check whether the existing loops are consistent. These algorithms essentially consist of a fixpoint computation that starts with unknown values for the output variables, and successively replaces these unknown values by known ones (which mimics the flow of information in a real hardware circuit). This analysis is usually done at compile time, although a dynamic analysis is also possible (see below). Problematic causality cycles may have different effects, which are illustrated with the help of some tiny examples below:

- The system described by the guarded actions \{\langle \text{x} \Rightarrow \text{x} = \text{true} \rangle, \langle \neg\text{x} \Rightarrow \text{x} = \text{false} \rangle\} is not deterministic; \text{x} can either be true or false. Similar to write conflicts, this problem would be hidden in the SystemC simulation, which would simply keep the old value of the variable.

- The system described by the guarded actions \{\langle \text{x} \Rightarrow \text{x} = \text{false} \rangle, \langle \neg\text{x} \Rightarrow \text{x} = \text{true} \rangle\} is not reactive. In contrast to the previous example, the SystemC simulation would also run into problems: The thread that is responsible for the computation of \text{x} would be triggered infinitely often and would change the value of \text{x} in each delta cycle. This behavior corresponds with an oscillation in hardware circuits, and this is exactly the behavior a synthesized circuit would show up. In consequence, simulation time does not advance for these kinds of programs.

- The system described by the guarded actions \{\langle \text{x}_1 \Rightarrow \text{x}_2 = \text{true} \rangle, \langle \neg\text{x}_1 \Rightarrow \text{x}_2 = \text{true} \rangle, \langle \text{x}_2 \Rightarrow \text{x}_1 = \text{true} \rangle\} is consistent (i.e. it has exactly one behavior), but it is not constructive. This means that the information whether \text{x}_2 is true or not cannot be inferred without guessing a value for \text{x}_2. For this concrete example, the SystemC simulation can compute the correct behavior (by accident), but in general, oscillations are possible, which prevent a progress of simulation time.
SC_MODULE (last3) {
  /∗ clock of the module ∗/
  sc_in_clk clk;
  /∗ inputs ∗/
  sc_in<int> i;
  sc_in<bool> sel;
  /∗ outputs ∗/
  sc_out<int> l1, l2, l3;
  /∗ local variables ∗/
  sc_signal<int> a1, a2, l1s, l2s, l3s;
  /∗ method declarations ∗/
  ...

  /∗ constructor ∗/
  SC_CTOR (last3) {
    /∗ threads ∗/
    SC_CTHREAD (control_flow, clk.neg () )
    SC_THREAD (compute_l1);
    sensitive << i << sel << l1s;    
    SC_THREAD (compute_l2);
    sensitive << sel << a1 << l2s;
    SC_THREAD (compute_l3);
    sensitive << sel << a2 << l3s;

    /∗ initialization ∗/
    a1.write (0);
    ...
  }
}

void last3 :: compute_l1 () {
  while (true) {
    if (sel.read ())
      l1.write (i.read ());
    else
      l1.write (l1s.read ());
    wait ();
  }
}

void last3 :: compute_l2 () {
  while (true) {
    if (sel.read ())
      l2.write (a1.read ());
    else
      l2.write (l2s.read ());
    wait ();
  }
}

void driver :: drive () {
  for (int j = 0; j < n; j++) {
    sel.write (sel_trace [j]);
    i.write (i_trace [j]);
    wait ();
  }
}

int sc_main(int argc, char* argv []) {
  sc_clock clock ("Clk",sc_time (1.0, SC_NS));
  sc_signal<int> i, l1, l2, l3;
  sc_signal<bool> sel;
  last3 last3_inst ("Last3");
  driver driver_inst ("Driver");

  /∗ connect signals ∗/
  last3_inst .clk (clock);
  ...

  sc_start (sc_time (14.0, SC_NS));
  return 0;
}

Figure 5: Example: Generated SystemC Code
To avoid the above problems, we check for causality problems and write conflicts before the translation. This does not require any additional effort, since all synchronous programs are usually checked at compile-time. Hence, we can guarantee that the generated SystemC components do not get stuck at a certain point of time.

In principle, it would be possible to check causality and write conflicts also at run-time, i.e. during the SystemC simulation. This can be simply accomplished by introducing additional variables that check whether a variable has been previously assigned at the same simulation time and imitating the fixpoint iteration which is implemented by compilers for synchronous languages. However, this doubles the computations, and therefore we see no added value in this approach, so that we do not follow it.

5 Summary

In this paper, we present a translation procedure from synchronous programs like Esterel and Quartz to discrete-event programs like SystemC. Starting from the representation as guarded actions, SystemC modules are generated that can be used within existing SystemC frameworks. We validated our transformation with several examples and outlined the consequences of write conflicts and causality problems for the generated SystemC program.

References


