Translating Concurrent Action Oriented Specifications to Synchronous Guarded Actions

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

Sandeep K. Shukla
FERMAT Lab
Electrical and Computer Engineering
Virginia Tech, Blackburg, VA, USA
http://www.fermat.ece.vt.edu/

Abstract

Concurrent Action-Oriented Specifications (CAOS) model the behavior of a synchronous hardware circuit as asynchronous guarded actions at an abstraction level higher than the Register Transfer Level (RTL). Previous approaches always considered the compilation of CAOS, which includes a transformation of the underlying model of computation and the scheduling of guarded actions per clock cycle, as a tightly integrated step. In this paper, we present a new compilation procedure, which separates these two tasks and translates CAOS models to synchronous guarded actions with an explicit interface to a scheduler. This separation of concerns has many advantages, including better analyses and integration of custom schedulers. Our method also generates assertions that each scheduler must obey that can be fulfilled by algorithms for scheduler synthesis like those developed in supervisory control. We present our translation procedure in detail and illustrate it by various examples. We also show that our method simplifies formal verification of hardware synthesized from CAOS specifications over previously known formal verification approaches.

Categories and Subject Descriptors B.5.2 [Hardware]; Register-Transfer-Level Implementation—design aids; B.6.3 [Hardware]; Logic Design—design aids; D.3.4 [Programming Languages]; Processors—compilers and code generation; D.3.2 [Programming Languages]: Language Classifications—concurrent, distributed, and parallel languages

General Terms Algorithms, Languages

Keywords Concurrent Action-Oriented Specifications, Guarded Commands, Synchronous Languages, Code Generation

1. Introduction

Guarded actions are a well-established concept for the description of concurrent systems. With their original theoretical background in term rewriting systems, they have been used in many specification and verification formalisms, e.g. Dijkstra’s guarded commands [7], Unity [5], Murϕ [8], DisCo [12], or the Temporal Logic of Actions [13].

In recent years, guarded actions have also found their way into hardware synthesis. Concurrent Action-Oriented Specifications [10] model the behavior of a hardware circuit as guarded actions at an abstraction level higher than the RTL. Thus, they aim at describing the data-flow of a hardware circuit without fixing its timing but only its causalities. Thereby, developers can defer the difficult task of global scheduling and coordination to the compiler. The Bluespec language and compilation̈ is based on these considerations, and it has proven to be very efficient for synthesizing hardware from CAOS models. It transforms the set of asynchronous guarded actions into a synchronous model by merging so-called conflict-tree transitions, which are executed (a.k.a. fired) in the same clock cycle.

Guarded actions are also very useful for the compilation of imperative synchronous languages like the Quartz language [4, 17, 19]. They are used as an intermediate code format, which abstracts from the various control-flow features and their complicated semantics, so that synthesis tools only have to read code with a simple semantics. While synchronous guarded actions have the same structure as guarded actions in all the other formalisms, they have a different underlying model of computation.

In general, a guarded action (which is often referred to as a rule in CAOS) has the form \((\gamma \Rightarrow C)\). The Boolean condition \(\gamma\) is called the guard and \(C\) is called the body of the guarded action, which usually modifies the variables over which the guards are defined. The intuition behind guarded actions is that the body is executed if the whole action is activated, i.e. if and only if its guard evaluates to true in the current state.

Usually, and in particular in CAOS, guarded actions are seen as an asynchronous model. In the current state, the guards of all actions are checked and subsequently, any one among the activated actions is selected and subsequently its body is executed. Inside the body, there are multiple statements which are considered to execute in parallel. We consider this model as an asynchronous one because there is no notion of synchronous execution of multiple rules in this reference semantics of CAOS. Rules execute one by one as their guards become true, and in the order the reference scheduler picks them. Note that the execution of behavior under this model of computation is also nondeterministic. As a result, a CAOS model under reference semantics is considered as a specification, and any possible execution behavior has to be implemented by a deterministic implementation. Hence, a deterministic implementation is one where a scheduler deterministically chooses the rules with certain predefined properties.

Synchronous guarded actions differ from that behavior: their model of computation does not only fire all activated actions, but...
the execution of the actions is postulated to happen simultaneously to the evaluation of the guards so that there may be interdependencies. In practice, this means that the execution follows the data dependencies between the actions. While this seems to be an unnatural model at first, this represents exactly the way a synchronous hardware circuit works: all computation and communication within a clock cycle happens more or less simultaneously according to the data dependencies. No synchrony assumption regarding execution time is required either in this execution model. Note that synchronous guarded actions provide a deterministic model, in the sense that all nondeterminism is pushed to the inputs. For a given input sequence, the execution sequence is determined, which is not the case for a CAOS model. This shows the distinction between the underlying models of computation. Therefore, the transformation of a CAOS model to a synchronous guarded action model requires to control the nondeterminism by some additional inputs. The stimuli to these inputs are then generated by schedulers. Since different schedulers may provide different stimuli that may not be compatible with the original CAOS model, the properties of acceptable schedulers must be provided by the transformation algorithm. Temporal assertions are one way to achieve this goal.

The contribution of this paper is the provision of a compilation procedure from CAOS to synchronous guarded actions. In contrast to previous work, which always considered the translation of the model of computation and the scheduling simultaneously, our approach separates these two tasks. From the original CAOS model, we do not construct one particular synchronous implementation, but we create synchronous guarded actions that cover all possible implementations. Since this model is deterministic, we encode the choice between all the possible schedules as additional inputs to the system which are constrained by generated assertions. Thus, linking a scheduler to the system which complies with the assertions results in one of the possible deterministic synchronous implementations that implement the original CAOS model.

In order to construct the scheduler constraints, we formally define the correctness preservation criterion, i.e. we fix the properties of the system which have to be obeyed by an implementation so that it is considered to be valid with respect to the original CAOS model. In principle, all previous definitions of conflict-free transitions had the same goal, but they were always made with consequences for a specific implementation in mind. Since we do not want to make any restrictions in advance, we aim at providing very general constraints, which are only implied by the CAOS semantics.

The separation of concerns does not only provide a better theoretical understanding of existing CAOS languages and synthesis tools such as Bluespec, but it also has many practical advantages: The representation as synchronous guarded actions allows us to carry out many analyses and optimizations known from the synchronous domain. In particular, specification and verification based on temporal logics becomes possible. Moreover, we can reuse synthesis tools for synchronous guarded actions (as provided by the Averest system\(^2\)), which support hardware and software as targets.

Since the scheduler is not compiled into the system, it is possible to verify an unscheduled CAOS model within this context, i.e. it is possible to prove system properties which do not depend on a concrete scheduler. From the theoretical side, it is even possible to create a scheduler according to a specified property using supervisory control [14, 27]. One can contrast this against [26] where verification always considers models that integrate data-flow and scheduling, even for properties that only depend on one part. In our approach, this can be separated, and substantially simplified. In addition, the framework makes it possible to create custom schedulers according to some requirements (e.g. latency, peak-power requirements) and link them to the translated model. With the help of the previously created assertions, the resulting system can be checked for correctness before synthesis. Thereby, custom schedulers can be generated by any tool and their correctness can be formally verified. Currently, developers cannot describe schedulers separately, they only have the possibility to encode them by compiler options or explicitly in the system model. Especially the last alternative is infeasible if several variants of a system are generated, which only differ in nonfunctional requirements reflected in the scheduler.

As far as we know, the closest paper to this is [6], which presented scheduling as rule composition. It gives a formal semantics of the execution of individual guarded actions and defines how the set of guarded actions can be composed to new derived ones. Scheduling is then defined to be the process of combining all guarded actions to a single one, which can be trivially translated into the synchronous domain. Our approach is different: We immediately switch to the synchronous domain, which also requires that we give a correctness preservation criterion which relates the model and its implementations. In addition, the scheduler is not mixed up with the data-path. It is kept separate until the final synthesis step which gives us all the advantages mentioned above.

This paper is structured as follows: Section 2 gives an overview of our starting point (Concurrent Action-Oriented Specifications). Section 3 subsequently highlights synchronous guarded actions (our target) and their underlying model of computation. In Section 4, we show our translation procedure in detail. In Section 5, we highlight how our approach can be advantageous for the formal verification of custom schedulers over the approach in [26]. In Section 6, we illustrate our approach by an example, before we conclude with a short summary in Section 7.

2. Concurrent Action-Oriented Specifications

2.1 Syntax

In this paper, Concurrent Action-Oriented Specifications (CAOS) are the starting point of our translation process. Instead of their full syntax and semantics, we focus on simple asynchronous guarded actions in the form of rules and methods, which are the language core and sufficient to define other statements as simple syntactic sugar. We first give an overview of the syntax before we describe their semantics in the next subsection.

Each CAOS model is defined over a set of explicitly declared variables \( V \), which represent the state of the modeled component. The behavior is described by a set of rules, which are guarded atomic actions of the form \( r_i : \text{when}(\gamma_i) \ S_i \), where \( \gamma_i \) is called the guard and \( S_i \) is called the body of rule \( r_i \).

\[
\text{rule } r_1 \text{ when}(\gamma_1) \ S_1 \\
\vdots \\
\text{rule } r_n \text{ when}(\gamma_n) \ S_n
\]

Provided that \( \sigma \) is a Boolean expression, \( \tau \) an expression of appropriate type over the readable variables, and \( x \) a writeable variable, the body \( S \) is one of the following statements:

- nothing: (no operation)
- \( x = \tau; \) (wire assignment)
- \( \text{next}(x) = \tau; \) (register assignment)
- \( S_1 \parallel S_2 \) (parallel composition)
- \( \text{if} (\sigma) \ S_1 \text{ else } S_2 \) (alternative)

CAOS provides two kinds of assignments: while wire assignments are immediately visible, register assignments are committed with the current state update. Other CAOS formalisms (like Bluespec) often distinguish these two variants by variable declarations,

\(^2\)http://www.averest.org
i.e. variables are declared either as wires or as registers. We use the more general approach in which appropriate assignment types distinguish between a state variable and a wire type variable. Several assignments can be combined in the body by parallel composition, which is simply written as the concatenation. Finally, alternatives in rules can be given by the if operator.

For the interaction with the environment, CAOS uses so-called methods, which are parameterized rules. In addition to the local variables, the action of a method has access to the variables specified in its parameter list. Traditionally, CAOS distinguishes so-called action methods, value methods and action-value methods: as the name suggests, an action method executes an action, which only transports data given by the parameters into the system, while a value method does not change the system state and simply returns a value — thus, they only transport data from the system to the outside. Action-value methods are just combinations of them. Frequently, it is required that the outputs of these methods do not depend on the inputs, since this can lead to cyclic dependencies. We do not make this assumption here, since our translation target (synchronous guarded actions) can deal with cyclic dependencies, and well-established analysis tools will spot problematic situations.

\[
\text{method } m_1(p_{11}, p_{12}, \ldots) \text{ when}(\gamma_1) S_1 \\
\vdots \\
\text{method } m_n(p_{n1}, p_{n2}, \ldots) \text{ when}(\gamma_n) S_n
\]

In the following, we do not use the names action methods, value methods and action-value methods, but we take simple parameter lists for our methods, which may contain inputs and outputs and thereby encode the appropriate variants of the method. In the concrete syntax, we use the prefix ? to mark a parameter as an input and ! to mark a parameter as an output.

2.2 Semantics

The CAOS semantics is very simple. After the initialization of all the variables, the following two steps are repeated forever: first, the guards of all actions are evaluated with respect to the current state. Among the actions whose guards evaluate to true, an arbitrary one is chosen and its body is executed. The execution generally modifies the system state so that other actions will be possibly activated in the following iteration. If no action is activated, the loop may be also aborted, since no state change will occur from there on.

Let \( q_0 \) be the initial state of the system, and \( q^0 \xrightarrow{S} q^1 \) denote that action \( S \) transforms the system in state \( q \) to state \( q^1 \). Then, a run of a CAOS model is a sequence of system states \( (q_0, q_1, \ldots) \) where \( q_i \quad q_{i+1} \) and \( \text{when}(\gamma_i) S_i \) is an arbitrary action which is activated in state \( q_i \), i.e. \( q_i(\gamma_i) = \text{true} \). Obviously, the system description is nondeterministic: even in the presence of the same inputs, which lead to the same activation of guards, the system can produce different outputs by choosing different activated actions. Hence, CAOS models are intended to be specifications, which describe a set of acceptable implementations.

Usually, one considers sequential hardware circuits as the implementation target for CAOS models. Thus, the compilation of a CAOS model involves a translation from one model of computation to another one, since the specification is asynchronous while the implementation is synchronous. Since the underlying model of computation has been changed, including the interface of the system, the communication has to be modified also: Input signals may change in each state without being considered by the system, because the rules that interact with them are not activated. Output signals must be also encoded in some way so that it is clear, what a single value is. In CAOS, methods are triggered by the environment of a module. They are commonly implemented by handshakes. The external module waits until the module can accept the request, then acknowledges it. Asynchronous behavior is thus implemented in synchronous one. Hence, the obvious question is: what properties have to preserved by the translation?

The correctness issue is addressed by the so-called reference implementation of a CAOS model. It maps all variables to hardware registers and each element (step) of the sequence to a clock cycle: i.e. in each step, it chooses exactly one action. Due to its inefficiency, the reference implementation is not considered as a real implementation. Instead, its main purpose is to define the correctness of real implementations. Since CAOS models are considered to be asynchronous, any implementation that respects the data dependencies given by the firing rules is by definition correct. Thus, schedulers usually try to fire as many activated actions as possible in a single clock cycle in order to produce an implementation with minimal latency.

Thereby, a real implementation only has to guarantee that it still complies with the CAOS semantics. This is guaranteed if the sequence of observed states is a subsequence of a run. Formally, to be considered correct, an implementation must always produce \( (q_0, q_1, q_2, \ldots) \) as its state sequence, such that there exist a run \( (q_0, q_1, q_2, \ldots) \) of the original CAOS model, where \( q_1, q_2, \ldots \in \mathbb{N} \) is an ascending subsequence of indices \( (0 < 1 < \ldots) \). Hence, if two actions are fired in the same step, the result must be the same as some sequence of them. Otherwise, this would not be a correct implementation.

In addition to this semantic constraint, there are more restrictions for merging actions: Executing the same CAOS methods (which are responsible for the interaction with the environment) more than once in a step, is not possible since simultaneous write accesses to the same interface signal would only preserve the last event (and simultaneous read accesses to the interface signal would replicate events). Similarly, a practical constraints is often given by available resources: Actions that share the same resource cannot be executed in the same clock cycle either. Thus, to save additional resources and to preserve traces, an implementation cannot arbitrarily merge a sequence of iterations into a single one, but only a sequence of iterations that is caused by different non-conflicting actions. Therefore, an implementation is assumed to fire a subset of actions of the CAOS model in each step, such that the choice of the subset guarantees the desired property by construction.

The CAOS compiler creates an implementation from a CAOS model by eliminating the nondeterminism from the original CAOS model according to the constraints presented above. It selects the action that is actually fired among the activated ones in dependence of the current inputs, and it merges a sequence of iterations into a clock cycle. These two tasks are handled by a scheduler, which is statically generated by the CAOS compiler. In each step, it dynamically selects among the activated actions the ones which are actually executed. To this end, it has access to the original guards \( \gamma_i \) of the actions and triggers their execution (see Figure 1). For the reference implementation, this scheduler consists of a simple priority encoder or a round-robin scheduler (if weak fairness is desired). A real implementation would try to select a somehow maximal set of activated actions according to the constraints explained above.

The overall synthesis approach is depicted in Figure 1. For all actions, hardware circuits to compute the next state is generated according to the corresponding right-hand sides \( \tau_i \) (lower cloud). Similarly, all guards \( \gamma_i \) are synthesized (upper cloud). They are used as input for the scheduler, which outputs the triggers for the actions \( \xi_i \). The actual execution of actions is implemented by a selector component, which chooses the values computed from the next state logic according to the trigger signals \( \xi_i \). These are finally used to update the system state for the following step.
3. Synchronous Guarded Actions

Synchronous guarded actions have the same syntactic appearance as the guarded actions of CAOS models. However, their execution is governed by the synchronous model of computation [1, 9] which is explained as follows: In general, the execution of a synchronous program is divided into macro steps that correspond with the interaction steps of a reactive system and its environment. Hence, for each macro step, new inputs are read from the environment, and new outputs and internal state changes are computed by the reactive system before the next interaction step takes place. To this end, macro steps are further divided into micro steps. Since the execution of micro steps is based on the same values of the input variables, their execution is viewed to be done in zero time, while the execution of a macro step requires one logical unit of time. In the programmer’s view, variables are therefore constant during the execution of a macro step requires one logical unit of time. In the programmer’s view, variables are therefore constant during the execution of a macro step.

The introduction of this logical time is not only the key for a straightforward translation of synchronous programs to hardware circuits [2, 15, 19]; it also provides a very convenient programming model, which allows compilers to generate deterministic single-threaded code from multi-threaded synchronous programs [3]. Thus, synchronous programs can be directly executed on simple micro-controllers without using complex operating systems. Furthermore, the concise formal semantics of synchronous languages makes them particularly attractive for reasoning about program properties and equivalences [18, 23].

The semantics of synchronous guarded actions implements the synchronous model of computation in the following way: In each macro step, all guards are simultaneously evaluated within the current variable environment. If a guard is true, its action is immediately executed. To this end, we have to distinguish between delayed (next(x) = τ) and immediate assignments (x = τ), which transfer the computed value to the left-hand side of the assignment in the next and current step, respectively. In the case of immediate assignments, the program may suffer from causality problems due to the immediate feedback.

To illustrate the effect of immediate feedback, consider the examples shown in Figure 2. 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. While these actions have no causality problem, the actions of the second column in Figure 2 represent a so-called causally incorrect program, which has no consistent behavior.

To avoid problems caused by causality cycles, compilers may check in the simplest case for absence of such cycles, or they can analyze the cycles by means of a causality analysis that guarantees that there exists an equivalent cycle-free system. This is a well-studied problem for synchronous systems and many analysis procedures have been developed to spot and eliminate these problems [20–22, 24].

Synchronous guarded actions without causality problems are always deterministic, because there is no choice among enabled guarded actions, since all of them 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 semantical problems.

To conclude this section, we claim that synchronous guarded actions are a very useful intermediate format for the compilation of synchronous languages, since they address a suitable level of abstraction: in particular, they provide a good balance between (1) removal of complex statements available at the source code level and (2) the independence of a specific synthesis target: On the one hand, complex control-flow statements like preemption statements have been eliminated during the translation to guarded actions. Hence, synthesis tools do not have to deal with the complex semantics of the statements and can focus on much simpler guarded actions. Despite their very simple structure, translation to both software and hardware is efficiently possible from guarded actions.

4. Translation

Translating CAOS to synchronous guarded actions gives rise to a fundamental problem: as CAOS is nondeterministic and our synchronous guarded actions are deterministic, we have to model the nondeterminism explicitly. To this end, we add additional inputs to the system, which are used to choose among the different alternative behaviors of the system. Intuitively, these additional inputs determine which activated actions are actually fired and when they are fired. In our translation, these new inputs will be provided by an explicit scheduler. Note that this scheduler is not part of the system itself, since we do not want to fix a concrete schedule. Instead, we enrich the system description by assertions, which are checked in all steps to preserve the CAOS semantics.

Thus, our design flow is as follows (see Figure 3): a CAOS model is translated to synchronous guarded actions with additional assertions. Then, a scheduler is written by hand or generated by some external tool according to the desired constraints and metrics (number of available resources, low latency, peak-power performance etc). After verifying that the assertions hold for the concrete scheduler, the guarded actions of the system and the scheduler are linked, constants are propagated and subexpressions are shared, and finally the synthesis procedures for synchronous guarded actions are invoked.

We implemented the translation presented in this section within the Averest system. This framework provides the basis for our approach by its various tools for synchronous systems. All its tools are built upon the common intermediate format AIF (Averest Intermediate Format), which is based on synchronous guarded actions and which can thus serve as the concrete synthesis target of our translation. Hence, the framework provides us with the tool support to implement and verify custom schedulers, to link all parts together and to finally synthesize hardware as well as software from synchronous guarded actions. For the implementation, we translate the CAOS primitives like rules and methods as presented in the rest of
Function CompileRule(\(\varphi, S\))

```
case S of
  nothing : return \{\};
  x = \tau : return \{\{\varphi \Rightarrow x = \tau\}\};
  next(x) = \tau : return \{\{\varphi \Rightarrow next(x) = \tau\}\};
  S_1, S_2 :
    R_1 = CompileRule(\(\varphi, S_1\));
    R_2 = CompileRule(\(\varphi, S_2\));
    return (R_1 \cup R_2);
  if(\(\sigma\)) S_1 else S_2 :
    R_3 = CompileRule(\(\varphi \land \sigma, S_1\));
    R_4 = CompileRule(\(\varphi \land \neg \sigma, S_2\));
    return (R_3 \cup R_4);
```

Figure 4. Preliminary Translation of Rules

This section into synchronous guarded actions. Then, the computation of the assertions is accomplished as described above, and both parts are finally assembled in an AIF file.

In this section, we provide more details about this translation: first, we show how the rules of the system are translated into the synchronous domain, before presenting the transformation of the methods in Section 4.2. In Section 4.3, we explain the construction of the assertions for the scheduling constraints, and finally Section 4.4 highlights confluence, an important property of CAOS models.

4.1 Rules

The rules of the CAOS model are defined over a set of variables. These are translated to so-called \texttt{m\textsc{emorized}} variables in the synchronous model. These variables store their values from one step to another, and they are mapped to registers in the final hardware synthesis step. In contrast, wires of the CAOS model become \texttt{event} variables in the synchronous model. These variables are reset to their default value if no action assigns them in the current step. Thereby, no registers to store the contained values are generated during the final synthesis for \texttt{event} variables.

The translation of the rules itself is given in Figure 4: for each rule \(\text{rule } r \text{ when}(\gamma, S_1)\), we declare an additional Boolean variable \(\xi_r\), which represents the trigger of \(r\) by the scheduler. Then, the guarded actions are simply translated into synchronous guarded actions by a call to \(\text{CompileRule}(\xi_r, S_1)\), which decomposes the body and creates for each assignment a synchronous guarded action. The new signal \(\xi_r\) as well as the original guard \(\gamma_r\) are exposed at the interface (see Figure 3) so that a scheduler can be connected, which drives the activation conditions depending on the activated guards. For example, the following rule \(R\)

```
rule R when(x > 0) {
  if(x = 42) {
    next(y) = y + 1;
    next(z) = true;
  } else
    next(y) = y - 1;
}
```

is translated to the following four guarded actions:

| \(\xi_R\) & \(x = 42\) & \(next(y) = y + 1\) & \(\xi_R \land (x = 42) \Rightarrow \text{next}(z) = \text{true}\) |
| \(\xi_R \land (x = 42)\) & \(next(y) = y - 1\) |

As already described above, the variable \(\gamma_R\) is declared as an additional output and \(\xi_R\) as an additional input, both connected to the scheduler of the system (see Figure 3).

4.2 Methods

CAOS uses methods to communicate with the environment. Unfortunately, they cannot be translated like all the other rules, because the interface of the system has changed from an asynchronous one to a synchronous one. This has some important consequences: the asynchronous CAOS interface implicitly guarantees that inputs are actually read by the system, since it assumes that methods are controlled by the environment. Thereby, the environment can be sure that its data has actually entered the system.

In contrast, the interface of module based on synchronous guarded actions just takes new input values in each step, no matter whether the system actually reads them or not. Thus, if a rule which is responsible for reading a value does not get activated in a step, the value at the interface is lost. The sender has no chance to detect this situation and to retransmit it, since there are no feedback signals in the opposite direction.

As a consequence, our translation has to generate the feedback explicitly, like any other CAOS compiler for synchronous target architectures. This can be accomplished by introducing a handshaking protocol for each method of the CAOS model, which ensures a correct interaction with the environment. In order to be compatible with previously synthesized circuits, we use the same interface as the Bluespec compiler, which implements this handshake by additional signals \(\text{rd}_m\) (method can be safely called) and \(\text{en}_m\) (invoke method call).
function CompileRule($\varphi, S$)

    case $S$ of
        nothing:
            return ($\{\},$true);
        $x = \tau$:
            return ($\{ (\varphi \Rightarrow x = \tau) \},$true);
        next($x$) = $\tau$:
            return ($\{ (\varphi \Rightarrow \text{next}(x) = \tau) \},$true);
        $m(x_1, \ldots, x_n)$:
            if $m$ has the signature method $m(p_0, \ldots, p_n)$
                $\mathcal{R} = \{ (\varphi \Rightarrow p_i = m_i \mid 1 \leq i \leq n) \}$
            return ($\{ (\varphi \Rightarrow \text{en}_m = \text{true}) \cup \mathcal{R} \},$true);
        $S_1, S_2$:
            $\mathcal{R}_1, \psi_1 = \text{CompileRule}(\varphi, S_1)$;
            $\mathcal{R}_2, \psi_2 = \text{CompileRule}(\varphi, S_2)$;
            $\mathcal{R}_1' = \{ \psi_1 \land \gamma \Rightarrow C \mid \gamma \Rightarrow C \in \mathcal{R}_1 \}$;
            $\mathcal{R}_2' = \{ \psi_1 \land \gamma \Rightarrow C \mid \gamma \Rightarrow C \in \mathcal{R}_2 \}$;
            return ($\mathcal{R}_1' \cup \mathcal{R}_2', \psi_1 \land \psi_2$);
        if($\sigma)$ $S_1$ else $S_2$:
            $\mathcal{R}_1, \psi_1 = \text{CompileRule}(\varphi \land \sigma, S_1)$;
            $\mathcal{R}_2, \psi_2 = \text{CompileRule}(\varphi \land \neg\sigma, S_2)$;
            return ($\mathcal{R}_1 \cup \mathcal{R}_2, \psi_1 \land \sigma \land \psi_2 \land \neg\sigma$);

Figure 5. Translation of Rules and Methods

Basically, the synchronous interface of a CAOS model is the collection of all its methods. Hence, we concatenate all parameter lists of the module to get the data interface of the synchronous module. Before this, we prefix all method parameters by the method names to avoid potential name clashes. In addition to this, the interface also contains control signals: for each method $m$, we introduce two Boolean signals of storage type event (i.e., they do not store values between clock cycles), an input $\text{en}_m$ and an output $\text{rdy}_m$. (In Figure 3 they are depicted by additional small arrows between the modules.) $\text{en}_m$ triggers method $m$ and tells the module that its input parameters are valid. $\text{rdy}_m$ tells the environment that method $m$ is ready to be called and that the provided output parameters are valid. The feedback via $\text{rdy}_m$ is necessary, since there may be situations, where methods cannot be called (e.g., due to the lack of required resources or in the course of an uncompleted computation). Otherwise, the module would have no chance to refuse a method call.

With the help of this extended interface, we can translate a method \text{method} $m(p_0, \ldots, p_n)$ when $\gamma$ \text{S} as follows: on the callee’s side, we generate synchronous guarded actions for the method almost like for ordinary rules (see Figure 4). The only difference is that we call the method trigger $\text{en}_m$, as guard for the actions, thus we do not expose the methods to the scheduler interface. In addition, we create the guarded action $(\gamma_m \Rightarrow \text{rdy}_m = \text{true})$, which drives the $\text{rdy}$ signal if the guard of the method holds. In contrast to ordinary rules, we do not expose the methods to the scheduler interface, since they are assumed to be controlled by the environment of the module.

On the caller’s side, we replace the method call by actions that set the trigger signal and the parameters of the corresponding method. Hence, we extend the function CompileRule($\varphi, S$) for method calls, i.e., the case $S = m(x_1, \ldots, x_n)$ as follows: For method $m$, we return the guarded actions $(\varphi \Rightarrow p_1 = x_1), \ldots, (\varphi \Rightarrow p_n = x_n)$ and $\varphi \Rightarrow \text{en}_m = \text{true}$. Additionally, we must handle the enabling conditions of the methods on the caller’s side by adding $\text{en}_m$ to the guards of all actions in the same alternative.

The revised version of the CompileRule functions (see Figure 5) implements the translation of the methods as described above. To handle the backward propagation of the method enabling conditions, it has two return values: the generated actions $\mathcal{R}$ and an additional enabling condition $\psi$ for parallel actions (in the same rule). When CompileRule is applied to a parallel composition $S_1 \parallel S_2$, the enabling conditions of one part is applied to the guards retrieved from the other part. When applied to an alternative if($\sigma)$ $S_1$ else $S_2$, both enabling conditions are combined with the help of the guard of the alternative $\sigma$.

The following code fragment illustrates the generation of synchronous guarded actions for methods. Assume that we have a rule $R$, which uses methods $M_1$, $M_2$ and $M_3$:

```plaintext
rule R when(x > 0) {
    M_1();
    if(x = 42) M_2(); else M_3();
}
method M_1() when(b){
    next(x) = 0;
}
```

Then, our translation extracts the following guarded actions for $R$ and $M_1$. While the guarded actions for the bodies are similar to the preliminary code generation for rules, the result for $\gamma_R$ is due to the backward propagation of the activation conditions of methods.

```
\[
\begin{align*}
\text{true} & \Rightarrow \gamma_R = (x > 0) \land \text{rdy}_{M_1} \land ((\text{rdy}_{M_2} \land (x = 42)) \lor (\text{rdy}_{M_3} \land \neg(x = 42))) \\
\xi_R & \Rightarrow \text{en}_{M_1} = \text{true} \\
\xi_R \land (x = 42) & \Rightarrow \text{en}_{M_2} = \text{true} \\
\xi_R \land \neg(x = 42) & \Rightarrow \text{en}_{M_3} = \text{true} \\
\text{true} & \Rightarrow \text{rdy}_{M_1} = b \\
\text{en}_{M_1} & \Rightarrow \text{next}(x) = 0
\end{align*}
\]
```

4.3 Scheduling Constraints

The synchronous guarded actions which have been constructed for a rule $r$ in the sections before, have been endowed by a completely new trigger $\xi_r$, which is not bound to the original guards $\gamma_r$ in the system description. This is accomplished by the external scheduler, which must ensure two basic properties: First, the scheduler must be safe, i.e., it should only fire actions that are activated. Thus, we add for each rule $r$ the assertion $\xi_r \rightarrow \gamma_r$. Second, the scheduler should preserve liveness, which is expressed by the LTL formula $\text{GF} \land \xi_r$ (where $\text{GF}$ means that $\Phi$ must hold in all cycles, and $\text{GF}$ means that $\Phi$ must hold eventually). Hence, for the rules $\mathcal{R}$ we generate the following assertions:

```
safety : assert (G \land_{r \in \mathcal{R}} \xi_r \rightarrow \gamma_r) \\
liveness : assert (\text{GF} \land_{r \in \mathcal{R}} \xi_r)
```

As already pointed out in Section 2, the CAOS semantics additionally requires that an implementation only executes a set of guarded actions simultaneously if they could have also been fired in a sequential order with the same result. So far, our synchronous system does not necessarily obey this rule and it can fire any set of actions depending on the original guards of the system. Since we do not want to fix any specific solution, we use synchronous assertions again to constrain the system behavior.

Obviously, all CAOS compilers have to solve a related problem. However, there is an important difference. We are not interested in an optimal schedule with respect to some property, but we want to give the set of all correct schedules. Nevertheless, we need to analyze the dependencies between the actions in the same manner. Thus, we first look at already existing definitions, e.g., the ones given by Hoe and Arvind in [10].
In essence, two rules can be fired simultaneously if the first rule does not have an impact on the activation of the second one in any state $q$ of the system. This is the underlying consideration of the following definition $r_1 \rightarrow r_2$ of sequentially composable rules:

$$r_1 \rightarrow r_2 \iff \forall q. (r_1 \neq r_2) \land [\gamma_1]_q \land [\gamma_2]_q \rightarrow [\gamma_2]|_{\delta_1(q)} \land \delta_2(\delta_1(q)) = \delta_1|_{\delta_2(q)}$$

Two rules $r_1$ and $r_2$ are considered to be sequentially composable ($r_1 \rightarrow r_2$) if the following property holds in all states: provided that the guards $\gamma_1$ and $\gamma_2$ hold in a state $q$, i.e. $[\gamma_1]_q \land [\gamma_2]_q$, then the guard of the second one still evaluates to true after firing the first one, i.e. $[\gamma_2]|_{\delta_1(q)}$, and the parallel firing of the bodies results to the same state as the sequential firing, i.e. $\delta_2(\delta_1(q)) = \delta_1|_{\delta_2(q)}$.

Obviously, a list of such rules $\langle r_1, \ldots, r_n \rangle$ such that two subsequent ones are sequentially composable, i.e. $r_1 \rightarrow \ldots \rightarrow r_n$, can be combined in a single step. In the synthesized circuit, this operation chains the next-state logic of all involved actions in exactly the same order. This is automatically accomplished by the synchronous synthesis tools, which considers the dependencies between all actions. As already shown in [10], the sequentially composable definition can be visualized by a directed graph whose edges represent dependencies between rules. In this so-called conflict graph, there are nodes for all rules and edges $(r_i, r_j)$ if the rules $r_i$ and $r_j$ do not satisfy $r_i \rightarrow r_j$.

The conflict graph is also important for our approach, since we can extract our assertions from it. Remember the goal is to find for any selected set of simultaneously fired actions a sequence of firing that has the same effect. Since the edges represent potential causal dependencies between actions, the sought order is just any one that does not contain any edge of the graph. If the actions are executed like this, no action can see the effect of any of its predecessors. Obviously, such an order can be always constructed if the selected set of actions does not contain a cycle of dependencies. All we need to extract the assertions is to find the cycles in the conflict graph: if a set of actions is fired which form a cycle in the graph, then none of them could have been fired first without making its effect visible to the others.

Hence, we add an assertion for each cycle in the graph, which is the conjunction of all rule triggers in the cycle. Thereby, our assertions guarantee that the scheduler never selects a set of mutually dependent rules. Formally, let $C = \text{cycles} \rightarrow$ be the set of nodes that form a cycle in the conflict graph for relation $\rightarrow$. Then we add for each cycle $C \in C$ in the graph an assertion of the following form:

$$\text{seqCompose}_C : \text{assert } (G \forall r \in C \xi_r)$$

In principle, cycles in the graph can be formed by mutually exclusive rules, which can be never activated simultaneously. Although we do not consider them separately, these assertions are quickly found by the verification tool with the help of the safety conditions given above.

Finally, the developer can add resource constraints given by his target architecture. All rules that use a shared resource can be set in conflict by additional assertions, which just forbid simultaneously executing the actions by setting corresponding $\xi$ trigger conditions.

From the theoretical side, this concludes the generation of assertions. However, large systems often make an extensive analysis of the conditions presented above very hard. While the safety property can be usually checked very efficiently, the liveness property and the conflict analysis can become very difficult. In particular, statically determining the sequentially composable relation is not simple\(^3\), conservative approximations can be used instead. By forbidding idle cycles, we can reformulate the liveness property as follows: in all steps, at least one trigger $\xi_r$ of the set of rules $R$ must hold, formally:

$$\text{liveness} : \text{assert } (G \forall r \in R \xi_r)$$

Similarly, we use a syntactical heuristic for the composition analysis, which just considers the read and write sets of the corresponding actions. This is given by the following definition of syntactically sequentially composable rules $r_1 \Rightarrow r_2$:

$$r_1 \Rightarrow r_2 \iff (r_1 = r_2) \lor (\text{grdVars}(r_2) \cap \text{rdVars}(r_2)) \cap \text{wrVars}(r_1) = \{\}$$

Thereby, grdVars $(a)$ is the set of variables occurring in the guard of $a$, rdVars $(a)$ is the set of variables read in the body, and wrVars $(a)$ is the set of variables written in the body. Again, the conflicts can be illustrated by a graph. A better alternative in this case is a bipartite graph, which contains a set of nodes for the variables and another set of nodes for the actions. However, the overall approach stays the same: we use $r_1 \Rightarrow r_2$ as a replacement for $r_1 \rightarrow r_2$, build its graph instead, determine all cycles and then add assertions accordingly:

$$\text{seqCompose}'_C : \text{assert } (\neg \forall r \in C \xi_r)$$

Since these definitions are not as accurate as the original ones, some correct schedulers may be rejected by the verification tool. However, the approximations scale better and therefore, they are the first choice for practical implementations.

Figure 6 shows a simple example. For the rules $r_1$, $r_2$, $r_3$, $r_4$, which are shown on the left-hand side, the conflict graph on the right-hand side can be retrieved. It has been constructed with the syntactically sequentially composable relation $\Rightarrow$, which only analyses the read and write sets of the rules. Two cycles can be identified: one containing the node set $\{r_1, r_3\}$ and another one containing the node set $\{r_2, r_4\}$. Thus, we add the following two assertions to constrain the synchronous guarded actions by the trigger signals $\xi_r$:

$$\left[\begin{array}{l}
\text{assert}(\neg(\xi_1 \land \xi_3)); \\
\text{assert}(\neg(\xi_2 \land \xi_4));
\end{array}\right]$$

4.4 Confluenve

In addition to pure correctness, which is described by the assertions of the previous section, developers are often interested in confluent

\(^3\)The problem becomes even more difficult if one does not use the set of all states but set of actually reachable states, which even depends on the concrete scheduler (which again depends on the sequentially composable relation).
ence. Since CAOS models are intended to be a specification and not a complete description of the system, they are nondeterministic. From the developers point of view, this nondeterminism is acceptable even if they have a single input-output behavior in mind, because the nondeterministic part may only affect the order in which actions fire and not the final observable result. In this case, the compiler can exploit the additional freedom and choose the optimal order (according to some criterion) among the possible ones. Formally, developers aiming to describe deterministic systems want confluence as known from term-rewriting systems.

Unfortunately, checking confluence is generally undecidable so that developers are responsible for guaranteeing this property. In particular, the CAOS semantics (and therefore the assertions presented in the previous section) do not require that a given model is confluent, and its compilers will not check for it.

What confluence basically says is that any permutation of a given set of rules applied sequentially leads to the same state as their concurrent execution. This is more restricted than sequentially composable, which only requires that the first rule does not have an impact on the activation of the second one (therefore, we could omit the assertions derived from the sequentially composable relation if we check for confluence). The conflict free relation given in [10] can be used for this task: it defines a bounded version of confluence, where the confluence point must be reached after executing each rule at most once. Obviously, two rules can be fired in any order if there are no dependencies between each other. This is the underlying consideration of the conflict free relation $r_1 \Rightarrow r_2$:

$$r_1 \Rightarrow r_2 \iff \forall q. (r_1 \neq r_2) \land \left[ \gamma_1 \downarrow q \land \gamma_2 \downarrow q \rightarrow \left[ \gamma_2 \downarrow \delta_1(q) \land \gamma_1 \downarrow \delta_2(q) \land \delta_2(\delta_1(q)) = \delta_1(\delta_2(q)) = \delta_1(\#2(q)) \right] \right]$$

Given a set of guarded rules such that they are pairwise conflict-free ($r_i \Rightarrow r_j$ for all $i$ and $j$), the rules can be executed independently from each other in the same step – in any order. In the generated hardware circuits, this has the effect that the next-state logic for all rules can be run in parallel and is not chained as in the case for sequentially composable rules so that its maximum latency is just the maximum of all fired rules and not the sum of them.

The conflict free definition can be visualized by a directed graph again: there, one adds for each pair of rules that does not satisfy $r_i \Rightarrow r_j$, the edges $(r_1, r_2)$ and $(r_2, r_1)$ to the graph. Extracting assertions from the graph follows exactly the same approach as described above for the sequentially-composable relation. Thus, we end up with the assertions of the following form (provided that $C \in \mathcal{C}$ is a cycle in the graph of $\Rightarrow$):

$$\text{locConfluent}_C : \text{assert } (G \neg J \land \bigwedge_{i \in C} \xi_i)$$

Again, statically determining the conflict free relation is a hard problem, so that there is an alternative syntactic check, which conservatively approximates the relation:

$$r_1 \Rightarrow r_2 \iff (r_1 \Rightarrow r_2) \lor \left[ \text{grdVars}(r_1) \cup \text{rdVars}(r_1) \right] \cap \text{wrVars}(r_2) = \{\} \land \left[ \text{grdVars}(r_2) \cup \text{rdVars}(r_2) \right] \cap \text{wrVars}(r_1) = \{\}$$

Unfortunately, this definition has shown to be very coarse and overly conservative. In many cases, schedulers are very restricted and cannot find a large enough set of rules that can be executed together in a cycle. Therefore, rules are only checked to be sequentially composable in CAOS designs.

5. Verification

The previous section presented the translation of CAOS models to synchronous guarded actions and assertions for the scheduler. As already mentioned in the introduction, separating the guarded actions and the scheduler is the basis for integration of custom schedulers. In this section, we illustrate how this makes the formal verification task substantially easier and simpler compared to previously presented techniques [26].

For instance, the synthesis of low-power hardware from CAOS models is considered in [25]. Its authors aim to reduce the peak power of a circuit generated from a CAOS model. Obviously, this nonfunctional requirement is mainly due to the scheduler of the system: the maximal set of rules that a scheduler selects in a given clock cycle directly influences the peak power of the overall circuit. Hence, they replace the scheduler that they have retrieved by their CAOS compiler by a custom one that is generated by an external tool according to the power requirements. One important problem then is to check whether the new scheduler complies with the original semantics of the CAOS model.

The authors try to answer this question in [26] in that they reduce the verification problem to a restricted form of automata language containment. Thus, each CAOS model is related to an automaton $A$, and its behavior is represented by the language of the automaton $L(A)$. Thereby, an implementation $R$ is considered to be correct if the language of its corresponding automaton is contained in the one of the original specification $S$: $L(A_R) \subseteq L(A_S)$. The whole process is implemented with the help of SPIN [11], where the automata are described as instances of PROMELA, the input language of SPIN. Since the current Bluespec compiler does not separate the scheduling and datapath during synthesis, the formal model built from the synthesized implementation contains all the complexities of the integrated datapath and the scheduler.

Hence, to verify that a given implementation complies to the original model, developers first synthesize the scheduled CAOS implementation model and generate a reference SPIN model for the whole system. Similarly, a SPIN model is created for the alternatively scheduled system. To this end, developers have to formalize the scheduler within the CAOS model, since there is no easy way for them to separate the datapath from the scheduler (the additional scheduler that the CAOS compiler creates for this is trivial). Finally, both SPIN models are used for the language containment check.

The advantages for carrying out formal verification using our approach over [26] can be summarized as follows:

- In the approach presented above, the alternative scheduler is intricately integrated into the CAOS model, which leads to a large description of the SPIN system, since it integrates the scheduler with the data path. In our approach, this data path is abstracted by the generated assertions. The actual computation of the data flow is not required in many cases.
- Since each scheduler integrated with the CAOS model represents a completely new hardware, the synthesis of the whole system needs to be repeated each time a scheduler is modified. In our approach, this data path is abstracted by the generated assertions. The actual computation of the data flow is not required in many cases.
- To check containment of the real implementation behavior in the reference implementation behavior set, intermediate steps from the reference model generally need to be hidden. This has shown to produce a considerable amount of additional variables in the models [26].
- Furthermore, we can use any temporal logic property checker for verifying the scheduler, since we do not depend on specific features of PROMELA or any other specification language. The assertions generated by our translation are simple LTL
module TokenRing {
    struct { int data; int addr; }
    Buf b1 = Buf();

event node1
    when((buf41.data != 0) & (buf41.addr == 1)) {
        next(buf12.data) = buf41.data;
        next(buf12.addr) = buf41.addr;
    }

event node2
    when((buf12.addr != 0) & (buf23.data == 0)) {
        next(buf23.data) = buf12.data;
        next(buf23.addr) = buf12.addr;
    }

event node3
    when((buf23.addr != 0) & (buf34.data == 0)) {
        next(buf34.data) = buf23.data;
        next(buf34.addr) = buf23.addr;
    }

event node4
    when((buf34.addr != 0) & (buf41.data == 0)) {
        next(buf41.data) = buf34.data;
        next(buf41.addr) = buf34.addr;
    }

    method send1(int ?a, int ?d)
    when(buf12.data == 0) {
        next(buf12.addr) = a;
        next(buf12.data) = d;
    }

    method receive1(int !d)
    when((buf41.data != 0) & (buf41.addr == 1)) {
        next(buf41.data) = 0;
        d = buf41.data;
        b1.ring();
    }
}

interface {
    bool ?xi_node1, bool !gamma_node1,
    bool ?xi_node2, bool !gamma_node2,
    bool ?xi_node3, bool !gamma_node3,
    bool ?xi_node4, bool !gamma_node4,
    bool ?en_send1, bool !rdy_send1,
    int !send1_a, int !send1_d,
    bool ?en_receive1, bool !rdy_receive1,
    int !receive1_d
}

main:
    buf12, buf23, buf34, buf41 = { 0, 0, 0, 0 };

    // --- rule node1 ---
    gamma_node1 = (buf41.addr != 1) & (buf12.data == 0);
    if xi_node1 : next(buf12.data) = buf41.data;
    if xi_node1 : next(buf12.data) = buf41.data;

    // --- rule node2 ---
    gamma_node2 = (buf12.addr != 2) & (buf23.data == 0);
    if xi_node2 : next(buf23.data) = buf12.data;
    if xi_node2 : next(buf23.data) = buf12.data;

    // --- rule node3 ---
    gamma_node3 = (buf23.addr != 3) & (buf34.data == 0);
    if xi_node3 : next(buf34.data) = buf23.data;
    if xi_node3 : next(buf34.data) = buf23.data;

    // --- rule node4 ---
    gamma_node4 = (buf34.addr != 4) & (buf41.data == 0);
    if xi_node4 : next(buf41.data) = buf34.data;
    if xi_node4 : next(buf41.data) = buf34.data;

    // --- methods ---
    rdy_send1 = (buf12.data == 0);
    if en_send1 : next(buf12.data) = send1_a;
    if en_send1 : next(buf12.data) = send1_d;
    rdy_receive1 =
        (buf41.data != 0) & (buf41.addr == 1) & b1.rdy_ring;
    if en_receive1 : d = buf41.data;
    if en_receive1 : next(buf41.data) = 0;
    if en_receive1 : d = b1.en_ring = true;

    // --- assertions ---
    assert !(xi_node1 & en_send1);
    assert !(xi_node1 & xi_node2 & xi_node3 & xi_node4);
}

Figure 7. CAOS Example and its Translation: Token Ring

formulas. To be precise, with the exception of the assertion liveness (see Section 4.3), all properties are even simple safety conditions, which can be verified by most model-checkers. In particular, language containment checks are not needed any more.
• Finally, our approach makes it possible to verify properties of the data path without supplying a scheduler at all. In this case, one uses the generated assertions as assumptions for the verification of the given properties.

6. Example
In this section, we illustrate our approach by a complete example. The CAOS model is given on the left-hand side of Figure 7, and the synchronous guarded actions derived from our translation on the right-hand side. The system describes a token ring, where messages can be only exchanged between neighbors by a common single-place buffer. Communication is directed and its direction is static, i.e., each buffer is always the input of the following node and the output of the preceding node.

In this example, we have four nodes connected to the ring, which all have the same behavior: if the output buffer is empty and the message in the input buffer is not for the node itself, the packet is forwarded. This part of the behavior is described by the rules node1, node2, node3 and node4. Packets are inserted and removed from the ring by send and receive methods. For the sake of simplicity, this example contains these methods only for the first node. By firing a send1, a new packet is inserted into the ring, which can be only done if the current output buffer of the first node is empty. Packets can be received by a call to receive1, which requires that there is a packet waiting in the input buffer of the node. If the packet is received, an external bell is rung by calling its ring method.

Obviously, all rules write to different buffers. The only resource conflict is between the forwarding of the first node and the introduction of a new packet, which both write to buf12. Hence, the
scheduler cannot fire them both, which would result to a write conflict. In the synchronous guarded actions, this is represented by the first assertion $\neg(\xi_{\text{node1}} \wedge c_{\text{hand1}})$. Another constraint is more interesting, which is due to the CAOS semantics. It forbids that all four nodes fire in parallel, since this cannot be represented by any sequential firing. This always requires that at least one buffer in the ring is empty in order to transmit a packet. Hence, one assertion is retrieved from the conflict graph, which describes exactly the apparent cycle: $\neg(\xi_{\text{node2}} \wedge \xi_{\text{node3}} \wedge \xi_{\text{node4}})$. Thereby, the scheduler can fire up to three arbitrary nodes if its aim is to produce maximum throughput. Another scheduler, which is optimized for low power, may only fire a single node in each step.

It is interesting to see that the case which has been identified by the analysis is not a critical one for the synchronous model. Due to its lockstep semantics, it allows the ring to forward all packets simultaneously, without the need for a bubble in the ring.

In the guarded actions given on the right-hand side of Figure 7, one may notice that many guards have a similar structure. This is a consequence of the translation, which prepends the activation condition of rule to all its actions. This could result to a quadratic blowup of the code size, which could quickly become a bottleneck for any practical translation. However, the actual intermediate code is more efficient: tables to store common terms are used to keep the size of the guarded actions linear to the CAOS model. This storage also eases a subsequent synthesis of the system.

7 Conclusions

In this paper, we presented a translation of CAOS models to synchronous guarded actions. While both descriptions are very similar to each other from the syntactical side, the main difference is the underlying model of computation. Hence, the main task of the translation is to bridge the gap between an untimed asynchronous description and a synchronous one. Thereby, scheduling constraints imposed by the CAOS semantics play a significant role. In contrast to previous approaches, we do not immediately include a scheduler in the synchronous model, but encode the constraints by additional synchronous assertions. This is not only a separation of concerns but it also pays off if custom schedulers should be connected to the system. They can be developed separately and integrated after the code generation of the main system. Additional assertions allow the developers to check whether the integrated scheduler complies to the CAOS semantics of the original module.

References