Round Trip to Asynchrony and Synchrony

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

Abstract
State of-the-art embedded system design methods generally use different languages and models, which differ in their abstraction levels and underlying models of computation (MoC). Depending on a particular application, one MoC may be better suited than another one, so that different MoCs are in use and have to be considered in the design of embedded systems. However, this heterogeneity makes an integrated analysis and synthesis of systems challenging. In particular, problems arise during synthesis if system parts have to be translated from one MoC to another one.

This paper considers two fundamental MoCs, which are based on different time models, namely synchronous and asynchronous (untimed) systems. We highlight their relationship and present transformations between these MoCs. We thereby abstract from concrete languages by using guarded actions to represent system behaviors.

1 Introduction
The development of efficient and reliable embedded systems is a difficult task. Various methodologies have been proposed over the last decades, among them model-based design, which integrates various design tasks like simulation, verification and synthesis (of both hardware and software) by using a set of models instead of particular implementations.

For the description of models, there is a large number of languages and formalisms available, ranging from discrete-event models such as SystemC [16, 22], synchronous languages [2, 13] such as Esterel [3], Quartz [27] or Lustre, to untimed formalisms such as SHIM [11] or static dataflow [19, 20]. All these languages focus on different aspects of the system, and they differ in their abstraction level and underlying model of computation (MoC) [12, 17, 21].

For example, the discrete-event MoC allows developers to describe systems very precisely and to simulate it efficiently. However, it is less suited for formal verification and synthesis. Synchronous MoCs are much better suited for formal verification and analysis due to their deterministic parallelism. Moreover, hardware synthesis for synchronous MoCs is straightforward, while software synthesis for multithreaded and distributed systems is challenging. In contrast, asynchronous systems can be efficiently compiled to these platforms but formal verification of real-world applications is often problematic due to nondeterministic parallelism and even worse state space explosion of asynchronous systems.

Hence, depending on a particular application or design phase, one MoC is better suited than another one. Since a heterogeneous embedded system is generally modeled in different MoCs, their integration [1, 12, 17, 21] has gained much interest in recent years. It is essential to obtain an integrated MoC which covers all used MoCs, so that the composed system can be simulated and analyzed. This integration is either based on a common meta-MoC, in which all integrated MoCs are implemented (e.g. tagged signal model [21], SystemMoC [14], ForSyDe [17]), or it is provided by
wrappers, which dynamically adapt the MoC to another one (e.g. latency insensitive design [6, 7] or endochronous systems [23]).

However, in order to fully benefit from model-based design, a simple integration is not enough. Usually, developers rely on predefined and highly optimized target architectures in order to meet common time and resource constraints. Thus, synthesis must map the models to target architectures, which might follow a different model of computation. In this case, neither common meta-models nor wrappers will help. Instead, the entire design must be translated, i.e. transformations from one MoC to another one are a key issue for synthesis.

In the following, we focus on two of the MoCs mentioned above, namely synchronous and asynchronous (untimed) MoCs. Synchronous systems are based on the notion of steps. In each step, they read inputs, update their state and produce outputs. These steps are atomic in the sense that synchronous systems do not describe any relationship between the actions that are executed in a step; they abstract from the causality, which is implicitly given by the data dependencies (e.g. one does not need to give an evaluation order for the gates in a synchronous circuit). In contrast, asynchronous systems do not have a predefined notion of time: they simply describe what actions follow another one, i.e. they precisely describe the causality. Hence, the synchronous and the asynchronous models focus on two different aspects: timing and causality.

The main contribution of this paper consists of the presentation of transformations between these two MoCs, which pave the way for a better exchangeability of synchronous and asynchronous models in a flexible model-based design flow. As the previous explanations suggest, the transformation from the synchronous to the asynchronous domain extracts causality, while the transformation in the opposite direction maps actions to steps according to their dependencies.

In the rest of this paper, we abstract from concrete programming languages, which differ in provided statements and description styles so that we do not dissipate our energy in language specific details. Instead, we build upon the very general model of guarded actions, a well-established concept for the description of concurrent systems. With a theoretical background in term rewriting systems, they have been used in many specification and verification formalisms (e.g. Dijkstra’s guarded commands [9], Unity [8], Murϕ [10] or DisCo [18]).

In recent years, guarded actions have also found their way into hardware synthesis. Concurrent Action-Oriented Specifications (CAOS) [15] model the behavior of a hardware circuit as guarded actions at an abstraction level higher than the RTL. The Bluespec system¹ is based on this model, and it has proven to be very efficient for synthesizing hardware from CAOS models. Guarded actions are also very useful for the compilation of imperative synchronous languages such as Quartz [5, 24]. They are used as an intermediate code format, which abstracts from the various statements and their complicated semantics so that synthesis tools are simplified.

¹http://www.bluespec.com/
The translations presented in the rest of this paper allow us to carry out many analyses and optimizations on asynchronous systems known from the synchronous domain. In particular, specification and verification based on temporal logics become possible. Similarly, it opens new possibilities for synthesis of synchronous languages. However, our ultimate goal is the translation between MoCs as shown in Figure 1 in system design: Source languages are first translated to an intermediate representation of the same MoC. At this level, the MoC can be changed by dedicated transformations, before the final synthesis accomplishes the generation of target code. This does not only separate concerns (by splitting up the translation into several parts) but it also has practical advantages. Creating the compiler infrastructure for a new MoC and new programming languages based on it is generally a hard and time-consuming task. In the domain of embedded systems, designing a compiler does not only require experts in the source language but also experts in analysis and in the target architectures. Hence, an intermediate level, which only deals with the translation of the MoC can bridge the gap between powerful programming language with complex semantics in MoC A and the low-level description of the target code in MoC B. Furthermore, an intermediate representation also allows designers to share common components of the compiler infrastructure: new input languages can be added by simply implementing the front-end, and new target architectures can be added by appropriate back-ends.

The rest of the paper is organized as follows: Our roundtrip will first highlight the synchronous MoC in Section 2 before we will visit the asynchronous MoC in Section 3. The following two sections show how models from one MoC can be translated to the other one. The first step in Section 4 translates synchronous to asynchronous MoCs, while Section 5 presents the converse translation. Finally, our round trip will end with a short summary in Section 6.

2 First Stop: Synchronous Systems

Synchronous systems [2, 13] are based on the notion of steps. 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. However, the most important aspect of synchrony is that all actions of a step are virtually executed at the same point of time. Well-known realizations are synchronous hardware circuits: all computations and communications within a clock cycle happen more or less simultaneously. In practice, this means that the execution within a cycle follows the data dependencies between the actions.

The systems we consider in the following are defined by sets of guarded actions. Each guarded action has the form \( \langle \gamma \Rightarrow A \rangle \), where the Boolean condition \( \gamma \) is called its guard and \( A \) its action. In synchronous systems, actions are either immediate assignments of the form \( x = \tau \) or delayed assignments of the form \( \text{next}(x) = \tau \). After the compilation of a program, we have for each writeable variable \( x \) of the system a set of immediate and delayed actions on \( x \):

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

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.

\[
\begin{align*}
a & \Rightarrow y = x + 1 \\
\neg a & \Rightarrow x = y + 2 \\
\text{true} & \Rightarrow z = x + y
\end{align*}
\]

\[
\begin{align*}
a & \Rightarrow x = \neg y \\
a & \Rightarrow y = x
\end{align*}
\]

\[
\begin{align*}
a & \Rightarrow x = \text{true} \\
a & \Rightarrow x = \text{false}
\end{align*}
\]

Figure 2: Synchronous Guarded Actions
Synchronous guarded actions are always deterministic, because there is no choice among activated
guarded actions. Instead, all of the activated actions must be fired. Hence, any system is guaranteed
to produce the same outputs for the same inputs. However, forcing conflicting actions to fire simulta-
neously leads to problems. To illustrate the effect of immediate feedback, consider the examples
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 has an immediate effect on 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
current macro step. Problematic cases are shown in the second and third columns of Figure 2. They
represent so-called causally incorrect programs, which have either no consistent or no deterministic
behavior. This is a well-studied problem for synchronous systems and many analysis procedures have
been developed to spot and eliminate these problems [4, 25, 26, 28, 29].

3 Second Stop: Asynchronous Systems

Asynchronous systems as considered here are defined over a set of declared variables \( V \), whose values
represent the state of the modeled system. Similar to synchronous systems, the behavior is described
by a set of guarded actions, which have the same form \( (\gamma \Rightarrow C) \) with a boolean guard \( \gamma \) and a body \( C \),
which usually modifies the variables over which the guards are defined. The intuition of asynchronous
guarded actions is that the body can be executed if its guard evaluates to true in the current state.

In our model, there are two different kinds of asynchronous guarded actions, namely rules and
methods. Rules are used to describe the internal behavior and have exactly the form as described
above. Methods are parameterized rules, which additionally have access to the variables specified in
their parameter lists. These parameters are used to model the communication with the environment.
In the rest of the paper, we use the following concrete syntax for rules and methods:

\[
\text{rule } r \ \text{when} (\gamma) : C \\
\text{method } m(p_1, p_2, \ldots) \ \text{when} (\gamma) : C
\]

The body of rules and methods \( C \) are sets of synchronous guarded actions. As expected, immediate
assignments are immediately visible, while delayed assignments are committed after the execution of
the whole body.

The semantics of the asynchronous guarded actions is rather simple. After the initialization of all
variables, the following two steps are repeated forever. First, the guards of all actions are evaluated
with respect to the current state to determine the set of activated actions. Then, an arbitrary activated
action 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 \( q \xrightarrow{S} q' \) denote that action \( S \) transforms the system in
state \( q \) to state \( q' \). Then, a run of a model is a sequence of system states \( \langle q_0, q_1, \ldots \rangle \) where \( q_i \xrightarrow{S} q_{i+1} \)
and \( \text{when} (\gamma_x) \ C_x \) is an arbitrary action which is activated in state \( q_i \), i.e. \( q_i(\gamma_x) = \text{true} \). Obviously,
asynchronous systems may be 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, models consisting of asynchronous guarded actions are often viewed as
specifications, which describe a set of acceptable implementations.

4 From Synchrony to Asynchrony

4.1 Causality Analysis

The translation from a synchronous model to an asynchronous one changes the underlying model of
computation of the system description. In particular, the asynchronous system needs to model the flow
of information, i.e., the causality, explicitly. For this purpose, our analysis introduces an additional flag \( kn_x \) for each writeable variable \( x \), which indicates whether the value of \( x \) is known or not. If \( kn_x \) is true, the value of \( x \) is the correct value for the current macro step. If \( kn_x \) if false, the variable \( x \) has an invalid value that must not be used.

Based on the flags \( kn_x \), we define a function \( kn(\phi) \), which describes whether \( \phi \) can be evaluated with the currently available information. Obviously, an expression \( \phi \) can be evaluated if all its free variables \( FV(\phi) \) are known. However, it can also be the case that its value can be determined without the knowledge of all variables: some operators may have dominant inputs that make others redundant, e.g., the second operand of \( a \land b \) does not need to be evaluated if \( a = \text{false} \). Lazy evaluation (see Figure 3) is based on this observation. In the following definitions, we use these rules to formally define \( kn(\phi) \) for an arbitrary formula \( \phi \):

- for variables and constants, we define
  \[
  kn(x) := \begin{cases} 
  \text{true} & \text{if } x \text{ is an input variable} \\
  kn_x & \text{otherwise}
  \end{cases}
  \]

- for boolean operators, we define:
  \[
  \begin{align*}
  kn(\neg \phi) & := kn(\phi) \\
  kn(\phi \land \psi) & := kn(\phi) \land kn(\psi) \lor kn(\phi) \land (\phi = \text{true}) \lor kn(\psi) \land (\psi = \text{true}) \\
  kn(\phi \lor \psi) & := kn(\phi) \lor kn(\psi) \lor kn(\phi) \land (\phi = \text{false}) \lor kn(\psi) \land (\psi = \text{false}) \\
  kn(\phi \rightarrow \psi) & := kn(\xi) \land kn(\phi) \land kn(\psi) \lor kn(\xi) \land (\xi = \text{true}) \land kn(\phi) \lor kn(\xi) \land (\xi = \text{false}) \land kn(\psi)
  \end{align*}
  \]

- for arithmetic operators, we define:
  \[
  \begin{align*}
  kn(\tau + \pi) & := kn(\tau) \land kn(\pi) \\
  kn(\tau - \pi) & := kn(\tau) \land kn(\pi) \\
  kn(\tau \times \pi) & := kn(\tau) \land kn(\pi) \lor kn(\tau) \land \tau = 0 \lor kn(\pi) \land \pi = 0 \\
  kn(\tau / \pi) & := kn(\tau) \land kn(\pi)
  \end{align*}
  \]

Based on this definition, we can describe the flow of information for a synchronous system given by a set of synchronous guarded actions of the form \( \langle \gamma \rightarrow x = \tau \rangle \). For each of these actions, we generate an asynchronous rule according to the following scheme:

\[
\text{rule } r_1 \text{ when } kn(\gamma) \land \gamma \land kn(\tau) \land \neg kn(x) : \\
\begin{align*}
  x & = \tau; \\
  kn_x & = \text{true};
\end{align*}
\]

If a synchronous system is causally correct, the rules will fire according to the data dependencies. If the system is not causally correct, all rules will be deactivated at some point of time so that no further progress will be observed.

For example, consider the following synchronous guarded actions. Note that they are cyclic (since \( x_1 \) and \( x_2 \) depend on each other) but causally correct.
Provided that \( a_1, a_2 \) and \( a_3 \) are inputs, applying the definitions above results to:

\[
\begin{align*}
\text{rule } r_1 & \text{ when } (a_1 \land (a_2 \lor \neg x_1)) : \\
x_1 &= \neg a_2 \land x_2; \\
\neg x_1 &= \text{true}; \\
\text{rule } r_2 & \text{ when } (a_3 \land \neg (a_2 \lor x_1)) : \\
x_2 &= a_2 \land x_1; \\
\neg x_2 &= \text{true}; \\
\text{rule } r_1 & \text{ when } (\neg x_1 \land \neg x_2) : \\
x_3 &= \text{true}; \\
\neg x_3 &= \text{true};
\end{align*}
\]

4.2 Interface

Another difference between both MoCs is their communication behavior at the system interface, which must be considered by the translation to preserve input-output behavior in a well defined way.

As already pointed out above, the synchronous semantics is implicitly based on the notion of (macro) steps, which include reading of inputs and writing of outputs. In contrast, the asynchronous system needs to pass the inputs and outputs explicitly by some kind of handshake protocol. Otherwise, input signals might change without being considered by the system (since the rules that read them are not activated), and a single output might be read several times by the environment.

Hence, when all variables have become known, the environment must be notified so that it can collect the outputs. If these outputs have been sent, the delayed actions can be executed, and new inputs for the following step can be set. Finally, the \( \neg x \) flag are reset so that a new step is triggered.

\[
\text{method } \text{step}() \text{ when } (\land \neg x) : \\
\text{read outputs */} \\
\text{execute delayed actions */} \\
\text{set inputs */} \\
\text{kn}_{x_1} = \text{false}; \ldots \text{kn}_{x_n} = \text{false};
\]

For our example, this is accomplished by the following rule:

\[
\text{method } \text{step}(i_1, i_2, i_3, o_1, o_2, o_3) \text{ when } (\neg x_1 \land \neg x_2 \land \neg x_3) : \\
o_1 = x_1; \ o_2 = x_2; \ o_3 = x_3; \\
a_1 = i_1; \ a_2 = i_2; \ a_3 = i_3; \\
\text{kn}_{x_1} = \text{false}; \text{kn}_{x_2} = \text{false}; \text{kn}_{x_3} = \text{false};
\]

5 From Asynchrony to Synchrony

5.1 Scheduling

Translating asynchronous to synchronous guarded actions gives rise to another fundamental problem: since the original asynchronous system does not include any timing information, there is no unique synchronous counterpart.

In order to define the correctness of the translation, we first consider the so-called synchronous reference implementation, which maps the execution of a single rule to a macro step. The translation has to guarantee that the sequence of observed states in the synchronous model is a subsequence of
a run of the asynchronous model. Formally, the synchronous model must produce \( \langle q_0, q_{i_1}, q_{i_2}, \ldots \rangle \) as its state sequence, such that there exist a run \( \langle q_0, q_1, \ldots \rangle \) of the original asynchronous model, where \( i_0, i_1, \ldots \in \mathbb{N} \) is an ascending subsequence of indices \( (i_0 < i_1 < \ldots) \). Hence, if two rules are fired in the same macro step, the result must be the same as some sequence of them.

In addition to this constraint, there are more restrictions for merging actions: Executing methods (which are responsible for the interaction with the environment) more than once in a macro 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, rules that share the same resource cannot be executed in the same macro step either. Thus, a synchronous system cannot arbitrarily merge a sequence of iterations into a single one, but only a sequence of iterations that is caused by different non-conflicting rules. Therefore, the synchronous model is assumed to fire a subset of actions of the asynchronous model in each step, such that the choice of the subset guarantees the desired property by construction.

To handle the scheduling, we add additional inputs to the system. Intuitively, these additional inputs determine which activated actions are actually fired and when they are fired. These new inputs will be later driven by an external scheduler. We do not integrate it in the system itself, since we do not want to fix a concrete schedule. Then, a scheduler can be added manually or by some external tool according to the given implementation constraints and metrics (number of available resources, low latency, peak-power performance etc). However, we enrich the system description by assertions, which constrain the set of possible schedulers that can be connected to the system.

The translation of the rules itself is done as follows: for each rule \( \langle \text{rule } r \text{ when}(\gamma_r) S_r \rangle \), we declare an additional Boolean variable \( \xi_r \), which represents the trigger of \( r \) driven by the scheduler. The new signal \( \xi_r \) as well as the original guard \( \gamma_r \) are exposed at the interface so that a scheduler can be connected, which drives the activation conditions depending on the activated guards. Then, \( \xi_r \) is added to the guards of all actions in the body.

For example, the following rule \( R \)

\[
\text{rule } R \text{ when}(x > 0) :
\begin{align*}
\text{true} & \Rightarrow x = \text{true}; \\
y = 42 & \Rightarrow \text{next}(z) = y + 1; \\
\neg(y = 42) & \Rightarrow \text{next}(z) = y - 1;
\end{align*}
\]

is translated to the following four guarded actions:

\[
\begin{bmatrix}
\text{true} & \Rightarrow \gamma_R = (x > 0) \\
\text{true} & \Rightarrow x = \text{true} \\
\xi_R \land (x = 42) & \Rightarrow \text{next}(y) = y + 1 \\
\xi_R \land \neg(x = 42) & \Rightarrow \text{next}(y) = y - 1
\end{bmatrix}
\]

### 5.2 Conflict Analysis

The synchronous guarded actions have been endowed by a completely new trigger \( \xi_r \), which is not yet bound to the original guards \( \gamma_r \). This is exactly the point where the external scheduler comes into play, which must ensure two basic properties when driving the new trigger. 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 formula \( \bigvee_r \xi_r \). Hence, for the rules \( R \), we generate the following assertions:

\[
\text{Safety} : \quad \text{assert} \left( \bigwedge_{r \in R} \xi_r \Rightarrow \gamma_r \right)
\]

\[
\text{Liveness} : \quad \text{assert} \left( \bigvee_{r \in R} \xi_r \right)
\]

As already pointed out above, the asynchronous 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 again synchronous assertions to constrain the system behavior.

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 [15]:

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

Two rules $r_1$ and $r_2$ are considered to be sequentially composable ($r_1 \rightarrow r_n$) 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. $\delta_2(\delta_1(q)) = \delta_1(\parallel(q))$.

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.

As already shown in [15], 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 $r_i \rightarrow r_j$. From this graph, we can directly extract our assertions. 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 desired 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 $C = \text{cycles}$ be the sets 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:

$$ \text{seqCompose}_C : \text{assert} (\lnot \bigwedge_{r \in C} \xi_r) $$

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 and liveness properties can be usually checked very efficiently, the conflict analysis can become very difficult. In particular, statically determining the *sequentially composable* relation is not simple. Instead, the following conservative approximations can be used, which just considers the read and write sets of the corresponding actions. This is given by the following definition of *syntactically sequentially composable* rules $r_1 \tilde{\rightarrow} r_2$:

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

Thereby, $\text{grdVars}(a)$ is the set of variables occurring in the guard of $a$, $\text{rdVars}(a)$ is the set of variables read in the body, and $\text{wrVars}(a)$ 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 remains the same: we use $r_1 \tilde{\rightarrow} r_2$ as a replacement for $r_1 \rightarrow r_2$, build its graph instead, determine all cycles and then add assertions accordingly. Since these definitions are not as accurate as the original ones, some correct schedulers may be rejected by the verification. However, the approximations scale better and therefore, they are the first choice for practical implementations.
rule $r_1$ when$(\gamma_1)$: $\text{next}(a) = b \land c$;
rule $r_2$ when$(\gamma_2)$: $\text{next}(b) = \text{false}$;
rule $r_3$ when$(\gamma_3)$: $\text{next}(c) = a \lor d$;
rule $r_4$ when$(\gamma_4)$: $\text{next}(d) = c$;

Figure 4: Conflict Graph Example

Figure 4 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 \textit{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 $\{r_3, r_4\}$. Thus, we add the following two assertions to constrain the synchronous guarded actions by the trigger signals $\xi_i$:

\[
\begin{align*}
\text{assert}(\neg(\xi_1 \land \xi_3)); \\
\text{assert}(\neg(\xi_3 \land \xi_4));
\end{align*}
\]

6 Summary

We considered two fundamental MoCs, namely synchronous and asynchronous (untimed) systems. For both MoCs, guarded actions can be used as adequate system representation. Although synchronous and asynchronous guarded actions have a similar structure, the underlying MoC gives them a completely different semantics: while the synchronous MoC focuses on timing, the asynchronous one focuses on causality.

As a first step towards a flexible MoC, we presented translations between the MoCs: asynchronous guarded actions are translated by a conflict analysis to synchronous guarded actions, while causality analysis is used for the opposite direction. These are the first steps towards a model-based design flow where synchronous and asynchronous models can both be used.

References


