From Clock-Driven to Data-Driven Models

Yu Bai, Klaus Schneider, Nikita Bhardwaj, Badarinath Katti, Tania Shazadi
University of Kaiserslautern
Kaiserslautern 67653, Germany
http://es.cs.uni-kl.de

Abstract—Clock/time-driven models are powerful abstractions of real-time systems, as e.g., provided by the synchronous models of computation which lend themselves well for simulation and verification. At every clock cycle, new inputs are read, computations are performed in zero-time, and results are immediately/synchronously communicated between components. However, such zero-time idealizations are not realistic since computation and communication finally takes time in implementations. For example, propagating clock signals synchronously became more and more difficult in circuit design and forced designers to consider elastic synchronous circuits [3]–[6] or even latency-insensitive designs [7], [8]. In embedded software design, the advent of multicore processors asks for multithreaded software systems whose communication is typically done via shared memory where synchronization is again very expensive.

In order to bridge the gap between synchronous models and asynchronous execution platforms, desynchronization methods have been proposed [9]–[20]. These approaches consider the synchronous system as a network of synchronous components which are first triggered by the same clocks. Desynchronization means then to remove these clocks so that the components have to decide by the arrival of input data whether they can execute a new execution step. It is known that this might introduce new behaviors in general, which is not the case for the subclass of endochronous components (in an isochronous network). Hence, if synchronous components were verified for endochrony, one can simply generate one thread per component to obtain a multi-threaded, distributed or asynchronous system that is completely driven by arrival of input data.

In this paper, we also consider a design flow that starts with a network of synchronous components. In this model, the system is considered at discrete points of time $t \in \mathbb{N}$ where each signal $x$ between two components may or may not be present at these points of time. Signals over a domain $D$ are therefore modeled as functions $\mathbb{N} \to D \cup \{\Box\}$, where $\Box$ is a dummy value that denotes that signal $x$ is absent at this point of time. Note that these values are only used for the synchronous modeling and are not used later in the asynchronous implementation. The use of values $\Box$ keeps the different signals aligned so that we can easily talk about synchronous reaction steps. For every signal $x$, one can define its clock $\text{clk}(x)$ which is of type $\mathbb{N} \to \{\text{true, false}\}$ such that for all points of time $\text{clk}(x) = \text{true}$ if $x \neq \Box$.

For an asynchronous implementation, we replace the synchronous network of components by an asynchronous network of components which can also be viewed as a dataflow process network [21]–[24]. To this end, values $\Box$ (or equivalently clock signals) are removed, and communication wires are replaced by FIFO buffers to allow input values to arrive at different points of time. While before this step, one value (possibly $\Box$) has to be read from each input buffer, it may now be the case that one or no value has to be read. The synchronous components are therefore enclosed by a wrapper that observes the arrival of the inputs and generates a local clock for the component to trigger its reaction. If for such a reaction, an input would have had value $\Box$ at the synchronous level, no value is consumed from the buffer, and instead, a dummy value $\Box$ is generated by the wrapper.

Since values $\Box$ are not used for computations, we can send any value to the inner synchronous component.

I. INTRODUCTION

The synchronous paradigm [1], [2] offers many advantages for the design of reactive systems. The key idea is the abstraction of time to discrete computation steps which are triggered by clocks of signals. Within each step, computations are considered to take place in zero time, and also communications are done instantaneously. Since variables may only be read after they have been assigned a value in each step, each variable has a precisely defined value in each step, which leads to a deterministic semantics. This simplified model of computation and communication allows designers to concentrate on the functional part of their designs.

While the synchronous paradigm lends itself well for modeling, simulation, and verification, its implementation on distributed or multi-threaded platforms is not efficient, since synchronization is very expensive on asynchronous platforms. For example, propagating clock signals synchronously became more and more difficult in circuit design and forced designers to consider elastic synchronous circuits [3]–[6] or even
In general, it is possible that a wrapper that preserves the functional behavior of the original synchronous system does not exist, and therefore one considers the class of endochronous systems. Intuitively, a component is endochronous if there is a unique way to consume the values of the input streams for computing the reactions.

Endochrony is however not sufficient to ensure the correct execution of synchronous components in an asynchronous way: It is possible that different endochronous components schedule the values of the signals to different reaction steps. Hence, one also has to consider isochrony: A network of components is isochronous if the components agree on synchronizing the asynchronous streams into synchronous reactions. In [25], we proved that each clock-consistent network of endochronous components is always isochronous. Clock-consistency can be checked by the compiler by static analysis techniques, which is out of the scope of this paper.

In this paper, we therefore concentrate on checking endochrony, which is the major requirement for desynchronizing a synchronous network of components\(^2\). For our correctness proofs, we describe the original synchronous component, its clock-scheduled variant, and its data-driven variant with symbolic transition relations that are well-known from model-checking. Figure 2 shows the variants and our procedure. We then prove that endochrony can simply be verified in that we check whether the data-driven component is still deterministic.

The outline of the paper is as follows: In the next section, we first discuss related work, and in Section III, we define the symbolic transition relation of a synchronous system that is defined by a set of guarded actions. In Section IV, we will then introduce symbolic descriptions of two other systems to prove the correctness of our endochrony criterion. Section V then provides experimental results, followed by the conclusions in the final section.

II. RELATED WORK

As mentioned in the introduction, desynchronization techniques have been proposed by researchers from both software and hardware domains. In circuit design, latency-insensitive systems and elastic circuits [7], [26] have been considered to cope with timing variabilities in synchronous circuit. In particular, elastic circuits can be viewed also as asynchronous circuits obtained from synchronous circuits. However except for [27], the main objective of the works did not focus on optimization of communication for multi-clocked systems. Practical latency-insensitive protocols still preserve all signal transmissions during communication. From a theoretical point of view, the concept of patient processes [7] shares a similar notion with endochronous data-driven components in this paper. However there is no detailed discussion further on checking when a process is patient. Aiming at multithreaded code synthesis for embedded software, we can not bear to preserve the full synchronous semantics of a clock-driven process because the penalty from communication overhead would be too heavy.

In the area of embedded software design, desynchronization methods can be dated back to works by Caspi and Girault [33] where the main purpose is to distribute a synchronous program. A detailed survey is provided in [20]. In general, we can roughly derive two categories for previous methods: automaton-based (earlier works) and clock-based (later works). Automaton-based methods compile programs into automaton-like intermediate codes and then distribute this code following the distributed data flow principle [29]. They mostly concentrate on distributing uni-clocked synchronous programs, and seldom considered multi-clocked programs. Later works [30], [31] take different speed of system components and therefore multilocated systems into consideration. In particular, for programs written in synchronous dataflow languages like Signal [12] and Lustre [32], it is possible to derive a clock tree describing clock relations between different system computations. Then a distribution based on clock relations can be derived, where least dependent components can be identified and distributed into different threads so that communication is optimized. This is much more difficult for imperative programs written in languages like Esterel [33] or Quartz [34], since clock relations are implicitly given by program semantics. Except for works by Girault et al. [30], no other works on desynchronizing imperative programs are clock-based. This work is originally designed for Lustre, therefore assumes that all programs to be distributed should possess a base clock (that is the finest clock for all signals) and will reject a program otherwise.

In clock-based methods, one problem is to check if absent values can be removed from communication. In our context, this is equivalent to check if a process can be transformed to a data-driven one without introducing nodeterminism. In [9], [15] the authors proved that it is sufficient to check properties like endochrony and weak endochrony. Checking endochrony is usually boiled down to check sequentiality, which is a sufficient condition. It is proved in [31] that a process is sequential if and only if its clock tree is single-rooted, i.e. there exists a base clock (which is trivial for Lustre, but not for Signal, as Signal programs allow a clock tree to be a forest). In [35] checking existence of a base clock is reduced to check if there exists a prime implicate of a boolean formula. Still, these methods are clock-based, and are not easy to be adapted for imperative synchronous programs. Instead, our method is based on clocked guarded actions [36] that is a common intermediate format for both imperative synchronous languages and dataflow languages. In particular, we reduced checking endochrony to an equivalent SAT problem, so that efficient symbolic model checking procedures can be applied. This distinguishes our work from all previous works.

III. SYMBOLIC REPRESENTATION OF SYNCHRONOUS SYSTEMS

In general, we describe a synchronous system \(P\) by a tuple \((\mathcal{V}, \mathcal{G})\) where the set of variables \(\mathcal{V}\) is a disjoint union \(\mathcal{V} = \mathcal{V}_{\text{in}} \cup \mathcal{V}_{\text{loc}} \cup \mathcal{V}_{\text{out}}\) of input, local and output variables, and \(\mathcal{G}\) is a finite set of guarded actions. Guarded actions are pairs \((\gamma, \alpha)\) where \(\alpha\) is an atomic action that is executed whenever its trigger condition (its guard) \(\gamma\) is true. Atomic actions are immediate assignments \(\lambda = \tau\) and delayed assignments \(\text{next}(\lambda) = \tau\) with a left-hand side expression \(\lambda\) and a right-hand side expression \(\tau\). Left-hand side expressions can be variables, or accesses to compound data types like tuples, arrays or records, and they always refer to an update of a single

\(^2\)In future work, we plan to also consider the verification of weak endochrony which allows desynchronization with a relaxed notion of correctness.
The symbolic description of valid behaviors of a variable \( x \in V_{\text{loc}} \cup V_{\text{out}} \) having clocked guarded actions \((\gamma_1, x = \tau_1), \ldots, (\gamma_m, x = \tau_m)\) is defined by \( R_x \) as follows:

\[
B_x \iff \bigvee_{i=1}^m \gamma_i, \quad T_x \iff \left( \bigwedge_{i=1}^m \gamma_i \implies x = \tau_i \right), \quad \mathcal{C}_x \iff (\text{clk}(x) = B_x), \quad R_x \iff B_x \land C_x \land T_x \quad \text{clock def.}
\]

The symbolic description of valid behaviors of a variable \( x \in V_{\text{loc}} \cup V_{\text{out}} \) having clocked guarded actions \((\gamma'_1, \text{next}(x) = \tau'_1), \ldots, (\gamma'_n, \text{next}(x) = \tau'_n)\) is defined by \( R_x \) as follows:

\[
B_x \iff \bigvee_{i=1}^m \gamma'_i, \quad T_x \iff \left( \bigwedge_{i=1}^m \gamma'_i \implies \text{next}(x) = \tau'_i \right), \quad \mathcal{C}_x \iff (\text{next}(\text{clk}(x)) = B_x), \quad R_x \iff B_x \land C_x \land T_x \quad \text{clock def.}
\]

Fig. 1: Transition Relation of a variable \( x \in V_{\text{loc}} \cup V_{\text{out}} \)

variable which is called the target variable. In the following, we may simplify this by only considering variables as left-hand side expressions.

The behavior of a synchronous system is straightforward: In every reaction step, all the guarded actions \((\gamma, \alpha)\) are executed whose guard condition \( \gamma \) is true. To this end, one has to make sure that there are no causality cycles, i.e., that the immediate actions \((\gamma, \lambda = \tau)\) are scheduled (executed in virtual micro steps) so that variables \( \lambda \) are only read if their value is ‘determined’ in this macro step. Causality analysis makes use of special values \( \perp \) to keep track of not yet determined values. In the following, we assume a causally correct synchronous system so that we do not have to deal with this problem. We also assume that there are no write conflicts, i.e., at every point of time, at most one value is assigned to a variable. Equally important, the behavior is undefined if none of the guarded actions is triggered. Therefore a synchronous system can reject some input if it does not satisfy any condition.

As shown in [34], we can moreover assume that for each variable \( x \in V_{\text{loc}} \cup V_{\text{out}} \) there are either only immediate assignments \((\gamma_1, x = \tau_1), \ldots, (\gamma_m, x = \tau_m)\) or only delayed assignments \((\gamma'_1, \text{next}(x) = \tau'_1), \ldots, (\gamma'_n, \text{next}(x) = \tau'_n)\). If that should not be the case, then one can introduce a new local variable \( x_c \) (called the carrier variable of \( x \)) that captures the delayed assignments, i.e., these are replaced with \((\gamma'_1, \text{next}(x_c) = \tau'_1), \ldots, (\gamma'_n, \text{next}(x_c) = \tau'_n)\) and adding the new guarded action \((\wedge_{i=1}^m \gamma_i, x = x_c)\). In the following, we can therefore say that a variable \( x \in V_{\text{loc}} \cup V_{\text{out}} \) is an immediate variable or a delayed variable (depending\(^3\) on whether it has immediate or delayed assignments).

Clearly, if a guarded action \((\gamma_i, x = \tau_i)\) is enabled\(^4\) then the current value of \( x \) is determined as the result of the evaluation of the right-hand side \( \tau_i \), and similarly the next value of \( x \) is determined if a guarded action \((\gamma'_i, \text{next}(x) = \tau'_i)\) is enabled. If neither is the case, the value of a variable \( x \) is determined by the reaction-to-absence which sets the clock of \( x \) to be false.

To encode the value \( \perp \), we introduce for every variable \( x \)

\(^3\)For \( x \in V_{\text{loc}} \cup V_{\text{out}} \) we assume that there is at least one assignment.

\(^4\)Guarded action \((\gamma_i, x = \tau_i)\) is enabled if \( \gamma_i \) is true.

its clock \( \text{clk}(x) \) as a corresponding boolean variable. Therefore, each clocked variable \( x \) is technically treated as a pair \((x, \text{clk}(x))\) where \( x \) carries the value of \( x \) and \( \text{clk}(x) \) carries the clock of \( x \). Using the clock signal \( \text{clk}(x) \), a clocked variable has value \( \perp \) if and only if \( \text{clk}(x) = \perp \) holds, and otherwise the value stored in \( x \) is the value of \( x \). Therefore, each condition \( \gamma \) can encode testing absence of an input \( x \) by testing \( \neg \text{clk}(x) \). The clock of a local or output variable can also be defined as shown in Figure 1. With the help of definitions in Figure 1, we can define the valid behaviors of a synchronous system as:

\[
R := (\bigwedge_{x \in V_{\text{loc}} \cup V_{\text{out}}} (T_x \land C_x) \land \bigvee_{x \in V_{\text{loc}} \cup V_{\text{out}}} B_x)
\]

and the transition relation as\(^5\) \( R || := (R \land \text{next}(R)) \). A state \( s \) assigns values to the variables \( V \) of the considered system, and \( s' \) assigns to the next variables \( V' \). Therefore \( R || \) relates two states \( s_1, s_2 \) whenever \( s_1 \land s_2' \) satisfies \( R \). We therefore write \( \{s_1, s_2\} \in R || \) if there is a transition between \( s_1 \) and \( s_2 \) and for a variable \( x \in V \), we write \( s_1(x) \) to denote its value in state \( s_1 \) (same for clocks \( s_1(\text{clk}(x)) \)).

The initial condition \( I_x \) of an immediate variable \( x \) is also its transition relation, i.e., \( I_x := R_x \). For delayed variables \( x \), we define \( I_x := \text{clk}(x) \land x = \text{default}(x) \), as if there is a previous cycle that assigned value to \( x \). The initial condition of the entire system is analogous, i.e., the transition relation the conjunction \( I || := \bigwedge_{x \in V_{\text{loc}} \cup V_{\text{out}}} I_x \).

For example, Figure 3 shows the Quartz code of a synchronous module that assigns its output \( y \) one of the inputs \( x2 \) or \( x3 \) depending on whether the first input \( x1 \) is true or not. It therefore has the set of variables: \( V_{\text{in}} = \{x1, x2, x3\} \), \( V_{\text{out}} = \{y\} \) and the following guarded actions:

- \( \text{clk}(x1) \land \text{clk}(x2) \land x1 \rightarrow y=(x2, \text{true}) \)
- \( \text{clk}(x1) \land \text{clk}(x3) \land x1 \rightarrow y=(x3, \text{true}) \)

The valid behavior \( R \) is therefore the conjunction of the following formulas:

- \( \text{clk}(y) = \text{clk}(x1) \land (\text{clk}(x2) \land x1 \rightarrow y=(x2, \text{true}) \land \text{clk}(x3) \land x1 \rightarrow y=(x3, \text{true})) \)

\(^5\text{next}(R ||) \) refers to the successive states of the transition system, where each variable \( x \) in \( R \) is replaced by its next version: \( \text{next}(x) \) in \( \text{next}(R) \).
module site(clock bool \{x1, x2, x3, y\}) {
    loop{
        if(clk(x1) & clk(x2) & x1) {
            y = (x2, true);
        }
        if(clk(x1) & clk(x3) & !x1) {
            y = (x3, true);
        }
        pause;
    }
}

Fig. 3: Quartz Code of Sequential If-then-else.
- clk(x1) & clk(x2) & x1 -> y=x2
- clk(x1) & clk(x3) & !x1 -> y=x3
- clk(x1) & (clk(x2) & x1 | clk(x3) & !x1)

In Figure 2(a) we show a sequence of inputs as well as a sequence of outputs, where each column corresponds to one cycle. For the first cycle, y is assigned (2, true) since \(x_1 = (true, true)\) and \(x_2\) is assigned to y.

IV. Desynchronization of Synchronous Systems

A. Stateless Components

We have already explained in the previous section that we consider synchronous systems \(\langle V, G \rangle\) given as a set of synchronous guarded actions \(G\) over a set of variables \(V\) that is a disjoint union \(V = V_{in} \cup V_{loc} \cup V_{out}\) over input, local and output variables. We have also explained that variables either have only immediate or only delayed assignments so that we speak about immediate and delayed variables, respectively. The latter transformation is very much related to the transformation of Mealy into Moore machines.

Synchronous systems can have an internal state that is determined by the previous values of the local variables. This internal state can influence the current reaction of a synchronous system, and a simple analysis of the transition relation defined in the previous section reveals that this can only happen if there are delayed variables.

For desynchronization, we define in this section a transformation of the system \(\langle V, G \rangle\) to a component of a dataflow process network (DPN). It is well-known that components of DPNs must not have an internal state, and therefore we first have to define a transformation that eliminates the potential internal state.

Once observed that the states of a DPN are encoded as the contents of the FIFO buffers between the components, this transformation is remarkably simple: Given a system \(\langle V, G \rangle\) with a delayed variable \(x \in V_{loc}\), we introduce a new variable \(x'\) and replace every guarded action \((\gamma, next(x) = \tau)\) with its corresponding immediate variant \((\gamma, x' = \tau)\). The new output \(x'\) of the component is written to a FIFO buffer where input values for variable \(x\) are read from. According to \(I_{x}\), this FIFO buffer must be initialized with the value \(\text{default}(x)\). In case \(x \in V_{out}\) holds, the FIFO buffer is read by the output port of \(x\) and everything else remains the same.

It is obvious that this way, we only have to consider synchronous components \(\langle V, G \rangle\) without internal state, and even without local variables. We therefore have components that are defined as follows:

**Definition 1 (Stateless Component \(R|\_\_\)\):** A synchronous component \(\langle V, G \rangle\) over variables \(V = V_{in} \cup V_{loc} \cup V_{out}\) with guarded actions \(G\) is called a stateless component if the following holds:

- \(V_{loc} = \{\}\)
- all guarded actions are immediate assignments

Therefore the valid behaviors of a stateless component defined by \(R\) contains no next variables. Also, since there is no delayed variable, the initial state \(I_{\_\_}\) becomes trivially true. Therefore the transition system is totally specified by \(R \land next(R)\). Since the only constraint over a transition \((s_1, s_2)\) is that both \(s_1\) and \(s_2\) should satisfy \(R\), the set of reachable states of \(R|\_\_\) are strongly connected. Since we assume causally correct systems, we can also rewrite these transition relations into equations of the form of Figure 4 that additionally makes clear that we deal with functions \(\bar{y} = f_c(\bar{x})\) from input values \(\bar{x}\) to output values \(\bar{y}\).

- \(\text{clk}(y_k) = \bigvee_{i=1}^m \gamma_i\) case
  - \(\gamma_i : \tau_i\)
- \(y_k = \begin{cases} \gamma_i : \tau_i \\ \vdots \\ \gamma_m : \tau_m \\ \text{else undefined} \end{cases}\)

Fig. 4: Function style representation of \(R\).

Notice that value \(\square\) is assigned to \(y\) once \(\text{clk}(y)\) is false no matter what the value of \(\tau_i\) is, therefore \(\text{clk}(y_k)\) and \(y_k\) both
Introducing buffers adds constraints to transition relation:
In particular, since clocks of variables are interpreted as instructions for consumption, a value should remain in the buffer until it is consumed. This is formally encoded as follows:

\[ \mathcal{R}_{buf} \Longleftrightarrow \mathcal{R}_\parallel \land \mathcal{BC} \]

using the following buffer constraint between two states \( s_1 \) and \( s_2 \):

\[ \mathcal{BC} \Longleftrightarrow \forall x \in \mathcal{V}_{in} . \ (\neg \text{clk}(x) \rightarrow \text{next}(x) = x) \]

Intuitively, the value \( x \) of an input variable now represents the first element of an input buffer, and the value of \( y \) of an output variable denotes the last element of that output buffer. If input \( x \) is not consumed, i.e., \( \text{clk}(x) \) is false, \( x \) is also the first element at the next point of time. Similarly, if \( \text{clk}(y) \) is false for output \( y \), then the last element must be the same as in the previous cycle. Notice that since \( \mathcal{V}_{in} \) and \( \mathcal{V}_{out} \) are disjoint, computations are totally determined by inputs, therefore we omit the buffer constraint for output variables.

One might wonder, whether \( \mathcal{R}_{buf} \) would lose some computations of \( \mathcal{R}_\parallel \) because of the additional constraint \( \mathcal{BC} \) (that eliminates some transitions of \( \mathcal{R}_\parallel \)). The following discussions show that this is not the case, and \( \mathcal{R}_{buf} \) and \( \mathcal{R}_\parallel \) are not really different in the sense that they still share the same computations. To this end, note that states \( s \) of the transition systems \( \mathcal{R}_\parallel \) and \( \mathcal{R}_{buf} := \mathcal{R}_\parallel \land \mathcal{BC} \) correspond with variable assignments, so that we write \( s(x) \) or \( s(\text{clk}(x)) \) to denote the values of \( x \) and \( \text{clk}(x) \) in state \( s \).

Definition 3 (Clock-Equivalence): Given states \( s_1 \) and \( s_2 \) of a transition system \( \mathcal{R} \) over variables \( \mathcal{V} \), we define for states \( s_1 \in \mathcal{S}_1, s_2 \in \mathcal{S}_2 \), the relation \( s_1 \approx_c s_2 \) as follows:

\[ s_1 \approx_c s_2 \iff \left( \forall x \in \mathcal{V} . \ s_1(\text{clk}(x)) = s_2(\text{clk}(x)) \land s_1(x) = s_2(x) \right) \]

Intuitively, \( s_1 \approx_c s_2 \) holds if the same values are present on \( s_1 \) and \( s_2 \), and moreover, for the present variables, \( x \) must be the same on \( s_1 \) and \( s_2 \). In other words, \( s_1 \approx_c s_2 \) allows only that variables \( x \) may have different values on \( s_1 \) and \( s_2 \) if these values are dummy values since the variable is actually absent \( \square \) (encoded by \( (x, \text{clk}(x)) \) with \( \text{clk}(x) = \text{false} \)).

Lemma 1 (Clock-Scheduled Component \( \mathcal{R}_{buf} \) of any component \( \mathcal{V}, \mathcal{G} \) with its transition relations \( \mathcal{R}_\parallel \) and \( \mathcal{R}_{buf} \), the following holds:

- For every path \( \pi : s_1, \ldots, s_n \) in \( \mathcal{R}_{buf} \), there is a path \( \pi' : s'_1, \ldots, s'_n \) of \( \mathcal{R}_\parallel \) such that \( s_i \approx_c s'_i \) holds for all \( i \in \{1, \ldots, n\} \).
- For every path \( \pi : s_1, \ldots, s_n \) in \( \mathcal{R}_\parallel \), there is a path \( \pi' : s'_1, \ldots, s'_n \) of \( \mathcal{R}_{buf} \) such that \( s_i \approx_c s'_i \) holds for all \( i \in \{1, \ldots, n\} \).

The corresponding paths in \( \mathcal{R}_{buf} \) and \( \mathcal{R}_\parallel \) are moreover uniquely defined.
Proof: Since $R_{buf}$ implies $R_v$, every transition of $R_{buf}$ is also a transition of $R_v$, and therefore any path through $R_{buf}$ is also a path through $R_v$. Therefore, the first item is immediately proved.

To prove the second item, consider a path $\pi: s_1, \ldots, s_n$ in $R_v$ and assume that $(s_i, s_{i+1}) \in R_v$ is the first transition on this path that violates the buffer constraint, i.e., $(s_i, s_{i+1}) \notin BC$. Now, we can define a new state $s_i'$ as follows: clocks are assigned in the same way as in $s_i$, i.e., $s_i'(clk(x)) := s_i(clk(x))$, and values are defined as follows: $s_i'(x) := s_i(x)$ if $s_i(clk(x)) = true$ and $s_i'(x) := s_{i+1}(x)$ if $s_i(clk(x)) = false$. By construction, we have $s_i \approx_c s_i'$ since we only modified variable’s values that are absent, and therefore still both $(s_i, s_i') \in R_v$ and $(s_i, s_{i+1}) \in R_v$ must hold (since both in $s_i$ and $s_i'$, each variable $x$ has the same value including $\Box$). Moreover, we now have $(s_i', s_{i+1}) \in BC$, since by construction $s_i(clk(x)) = false$ implies $s_i'(x) = s_{i+1}(x)$. This way, we now have a new path through $\pi: s_1, \ldots, s_{i-1}, s_i', s_{i+1}, \ldots, s_n$ in $R_v$ with $(s_i', s_{i+1}) \in BC$.

However, we now might have $(s_{i-1}, s_i') \notin BC$ due to our changes of $s_i'$. Hence, we also modify $s_{i-1}$ to a new state $s_{i-1}'$ by changing $s_{i-1}'(x) := s_{i-2}(x)$ whenever $s_{i-2}(clk(x)) \neq false$. This can be continued until either a state $s_{i-k}$ is found where $(s_{i-k}, s_{i-k+1}) \in BC$ holds, or until we worked through the path modifying all states up to the initial state $s_0$.

After this, the procedure has to reconsider the generated path that now satisfies the buffer constraints up to $(s_i', s_{i+1})$ and may have to correct another one after that transition in the same way as before.

The above lemma is quite interesting since it ensures that no computation is essentially lost in the sense that for each computation of $R_v$, there is a clock-equivalent computation in $R_{buf}$ and vice versa. The difference between $R_v$ and $R_{buf}$ is roughly explained as follows: consider a path $s_1, \ldots, s_n$ in $R_v$ where a variable $x$ is consumed in the final state $s_n$ only. Since input values are not buffered in $R_v$, we can assign variable $x$ arbitrary values on the other states $s_1, \ldots, s_{n-1}$ where $x$ is not consumed. Indeed, they may correspond to some output values from other components, where their clocks are set to false, and since $clk(x) = false$ means the data is actually absent $\Box$, any value is allowed. In $R_{buf}$, however, we have to set the value $s_i(x) := s_n(x)$ for all states on the path, which is not really relevant since the clock $s_i(clk(x))$ is false on these states. However, this satisfies the buffer constraint, and it means that the values that will be finally consumed should arrive after the previous consumption and are buffered until they were consumed (which models a FIFO buffer).

It can moreover be shown that $R_v$ and $R_{buf}$ are however not bisimilar, but only language equivalent. However, the quotient of $R_v \approx_c$ defines the real computations of $R_v$, and by Lemma 1, $R_{buf}$ contains exactly the same computations as $R_v$, and it is easy to see that their quotients by $\approx_c$ are the same.

From a functional point of view, clock-scheduled components only make clocks of variables explicit so that we can write: $f(y, clk(y)) = f(x, clk(x))$. For example, in Figure 2(b) we show the clock-scheduled component derived from site. Additional to data buffers of the variables, each variable now has a channel communicating the clocks. For the inputs, they are shown on the top of the component, and for the output $y$ it is below data channel $y$. The schedule instruction corresponding to the first reaction of Figure 2(a) is therefore $(clk(x_1) = true, clk(x_2) = true, clk(x_3) = false)$. The same output 2 is assigned to $y$, and an additional clock true is produced for $clk(y)$ as well. The values $x_1 = 0, x_3 = 3$ are now following directly after the data values from the first cycle, and no more $\Box$ is communicated anymore. Interestingly, our state transition system $R_{buf}$ remains a finite state system for finite data types. For site, $R_{buf}$ corresponds to the scissored part shown in Figure 2.

C. Data-Driven Processes

The system developed in the previous section is already an asynchronous version in the sense that the components can already fire their reaction steps independent of each other. However, it still makes use of the clock information $clk(x)$ to know whether a value from input buffer $x$ has to be consumed for the current reaction. In this section, we now remove the communication and computation of signals $clk(x)$, so that the components have to decide on their own whether there are enough input values to trigger a reaction. A component is called endochronous if for all input streams, there is a unique way to consume them while producing the related output streams.

We call a process without clock inputs $clk(x)$ a data-driven component. Given a synchronous component $P$, we denote its data-driven version by $P_{DPN}$. Its transition relation $R_{DPN}$ is defined as follows:

Definition 4 (Data-Driven Component $R_{DPN}$): Given a synchronous component $(V, G)$ over variables $V = V_{in} \cup V_{out}$ with only immediate guarded actions $(\gamma, x = \tau)$. Using the transition relations $R_v$ as defined in Figure 1, and $R_{buf}$ of Definition 2, we define the transition relation $R_{DPN}$ of the data-driven component as follows:

$$R_{DPN} := \exists ! clk(x), clk(x'). R_{buf} := \exists ! clk(x), clk(x'), R_v \land BC$$

where $x'$ refers to next($x$). Note that states $s_1$ and $s_2$ with $s_1(\Box) = s_2(\Box)$ but with $s_1(clk(x)) \neq s_2(clk(x))$ are merged by existential quantification. In general, $P_{DPN}$ might therefore no longer implement a function.

$$(clk(x_1) \& clk(x_2) \& clk(x_3) \& x_1 \& x_2 \& x_3) \Rightarrow y = (true, true)$$

$$(clk(x_1) \& clk(x_2) \& clk(x_3) \& x_1 \& x_2 \& x_3) \Rightarrow y = (true, true)$$

$$(clk(x_1) \& clk(x_2) \& clk(x_3) \& x_1 \& x_2 \& x_3) \Rightarrow y = (false, true)$$

Fig. 6: Guarded Actions of Component $P_{DPN}$
∀ input variables, i.e., we get justifies the relationship between $R_{\text{buf}}$ and $R_{\text{DPN}}$.

Definition 5 (Value-Equivalence): Given states $s_1$ and $s_2$ of a transition system $R$ over variables $\mathcal{V}$, we define for states $s_1 \in S_1, s_2 \in S_2$, the relation $R \approx \equiv s_1 \approx_v s_2$ as follows:

$$s_1 \approx_v s_2 \iff \forall x \in \mathcal{V} . s_1(x) = s_2(x)$$

Note that $R_{\text{DPN}}$ can also be defined alternatively as follows using $\approx_v$:

$$(s_1, s_2) \in R_{\text{DPN}} \iff \exists s'_1, s'_2 : (s'_1, s'_2) \in R_{\text{buf}} \land s_1 \approx_v s'_1 \land s_2 \approx_v s'_2$$

We denote $s|\mathcal{V}_m$ as the value assignment of $s$ projected to input variables, i.e., $\forall x \in \mathcal{V}_m . s|\mathcal{V}_m(x) = s(x)$. Given two paths $\pi_1 = s_1, s_2, \ldots, s_n \approx_v s'_1, s'_2, \ldots, s'_n$, we denote $\pi_1|\mathcal{V}_m \approx_v \pi_2|\mathcal{V}_m$ if and only if each $s_i|\mathcal{V}_m \approx_v s'_i|\mathcal{V}_m$. Intuitively, it means $\pi_1, \pi_2$ share the same sequence of input values.

Lemma 2 (Data-Driven Component $R_{\text{DPN}}$): For any stateless component $(\mathcal{V}, \mathcal{G})$ with its transition relations $R_{\text{buf}}$ and $R_{\text{DPN}}$, the following holds:

- For every path $\pi : s_1, \ldots, s_n$ in $R_{\text{buf}}$, there is a path $\pi' : s'_1, \ldots, s'_n$ of $R_{\text{DPN}}$ such that $s_1 \approx_v s'_1$ holds for all $i \in \{1, \ldots, n\}$.
- For every path $\pi : s_1, \ldots, s_n$ in $R_{\text{DPN}}$, there is a path $\pi' : s'_1, \ldots, s'_n$ of $R_{\text{buf}}$ such that $s_1 \approx_v s'_1$ holds for all $i \in \{1, \ldots, n\}$.
- The relation between $\pi$ in $R_{\text{buf}}$ and $\pi'$ in $R_{\text{DPN}}$ such that $\pi|\mathcal{V}_m = \pi'|\mathcal{V}_m$ is not a bijection.

Proof: Similar to Lemma 1, the first proposition is trivial: Given a path $\pi : s_1, \ldots, s_n$ in $R_{\text{buf}}$, all we have to do is to restrict the domain of the states\(^6\) $s_1$ to variables $x \in \mathcal{V}$ only to obtain the path in $R_{\text{DPN}}$.

To prove the second item, we first prove the following fact: (**) for all states $s_1, s_2, s_3$ with (1) $s_1, s_2 \in R_{\text{buf}}$ and (2) $s_2 \approx_v s_3$, we also have (3) $s_1, s_3 \in R_{\text{buf}}$.

By definition of $R_{\text{buf}}$, we derive from (1) $(s_1, s_2) \in R_{\text{buf}}$ that (4) $(s_1, s_2) \in \mathcal{R}$ and (5) $(s_1, s_2) \in R_{\text{buf}}$.

By the definition of $R_{\text{DPN}}$, we get (6) $\forall x \in \mathcal{V}_m . -s_1(\text{clk}(x)) \rightarrow s_2(x) = s_1(x)$, and since (2) means that (7) $\forall x \in \mathcal{V} . s_2(x) = s_3(x)$, we also have (8) $\forall x \in \mathcal{V}_m . -s_1(\text{clk}(x)) \rightarrow s_3(x) = s_2(x)$, which

\(^6\)Recall that states in $R_{\text{buf}}$ are variable assignments that map variables $x \in \mathcal{V}$ as well as their clocks $\text{clk}(x)$ to corresponding values.

The transition system $R_{\text{buf}}$ contains eight states, and we omit it due to lack of space. The behavior of $\text{alt}$ is quite easy. At each cycle, only one of the inputs $x_1, x_2$ should be present, and it is copied to the output $y$. Now consider the sequence of inputs with $x_1 : 1, 1, \ldots, x_2 : 0, 0, \ldots, x_1$. We can find two paths in $R_{\text{buf}}$ with the same sequence of inputs:

$$\pi_1 : x_1 : 1 \quad x_2 : 0 \quad y = 1$$

It is easy to verify that both paths exist in $R_{\text{buf}}$, and by item 1 we just proved, there are two corresponding paths $\pi'_1, \pi'_2$ in $R_{\text{DPN}}$. All these four paths share the same input sequence, therefore there is no bijection between $\pi'_1, \pi'_2$ and $\pi_1, \pi_2$.

Hence, similar to Lemma 1, the above lemma ensures that no computation is lost in $R_{\text{DPN}}$: just the clock information is removed from that computations in $R_{\text{DPN}}$. Hence, the computations of $R_{\text{buf}}$ and $R_{\text{DPN}}$ are the same in this sense. While there is a unique correspondence between computations in $R_{\text{buf}}$ and $R_{\text{DPN}}$, this is however not the case for the paths of $R_{\text{buf}}$ and $R_{\text{DPN}}$: Existential quantification over the clocks to derive $R_{\text{DPN}}$ from $R_{\text{buf}}$ merges states $s_1$ and $s_2$ with $s_1(\overline{x}) = s_2(\overline{x})$ but with $s_1(\text{clk}(\overline{x})) \neq s_2(\text{clk}(\overline{x}))$. Therefore, the transitions of these states are also inherited by the obtained merged state, which can introduce nondeterminism in $R_{\text{DPN}}$ that did not exist in $R_{\text{buf}}$. It is then not possible to derive a unique clock information since we could either add the clocks of $s_1$ or of $s_2$. For example, as revealed by component $\text{alt}$, the sequence of inputs in $R_{\text{DPN}}$ is able to derive two different sequences of outputs only by using different schedule instructions.

For this reason, a special class of synchronous systems has been introduced in [9]. For these systems, a unique addition...
of clock information on a path in $R_{DPN}$ exists to derive a path in $R_{buf}$ and therefore in $R_{ij}$. We give a definition of these systems in our setting and a criterion to check this property in the next subsection.

### D. Checking Endochrony

Given a stateless synchronous component $R_{ij}$, we have now introduced a buffered version $R_{buf}$ and an asynchronous data-driven version $R_{DPN}$ such that no computation of $R_{ij}$ is lost, neither in $R_{buf}$ nor in $R_{DPN}$, nor are new computations added. However, the addition of clock information is not always uniquely possible, so that several synchronous computations of $R_{ij}$ might be mapped to the same one in $R_{DPN}$. We therefore define endochronous systems as follows:

**Definition 6 (Endochrony):** A stateless component $(V,G)$ with its transition system $R_{buf}$ is endochronous if the following holds:

$$\forall s_1, s_2 \in R_{buf}. \forall t \in V_{in}. \quad s_1(x) = s_2(x) \rightarrow s_1(clk(x)) = s_2(clk(x))$$

Definition 6 is defined in the same spirit as [31]. Hence, a stateless component is endochronous, if the clock information is uniquely defined for all states $s_1 \approx_s s_2$. This means that adding clock information on a state in $R_{DPN}$ is possible only in an unique way, and therefore the paths of $R_{ij}$, $R_{buf}$, and $R_{DPN}$ have unique correspondences. We therefore immediately obtain our following main result:

**Theorem 1 (Endochrony):** The transition systems $R_{ij}$, $R_{buf}$, and $R_{DPN}$ for a stateless synchronous component $(V,G)$ have uniquely corresponding computation paths if and only if $(V,G)$ is endochronous.

**Proof:** The unique correspondence between computations in $R_{ij}$ and $R_{buf}$ is provided by the fact that they share the same quotient structure over $\approx_s$, proven by Lemma 1. To prove the unique correspondence between computations in $R_{buf}$ and $R_{DPN}$, we prove that the two transition systems are isomorphic by $\approx_s$, if and only if $R_{buf}$ is endochronous.

($\Rightarrow$) Assume we are given an endochronous component. If $R_{buf}$ and $R_{DPN}$ are not isomorphic, then by Definition 4 the only possibility is that there exist at least two states $s_1, s_2$ in $R_{buf}$ that are mapped to state $s'_1$ in $R_{DPN}$. This can only happen if that $s_1(x) = s_2(x)$ but $s_1(clk(x)) \neq s_2(clk(x))$ holds (since otherwise they are the same state), which contradicts our assumption.

($\Leftarrow$) Assume each state $s$ of $R_{buf}$ is mapped to a unique state $s'$ of $R_{DPN}$ by $\approx_s$, then it is trivial that there is no pair of states that violates the definition of endochrony, otherwise it contradicts the uniqueness.

Note that endochrony ensures determinism of $R_{DPN}$, i.e.,

$$\forall s_1, s_2 \in R_{DPN}. \quad (\forall x \in V_{in}. s_1(x) = s_2(x)) \rightarrow (\forall y \in V_{out}. s_1(y) = s_2(y))$$

This is explained as follows: If $s_1(x) = s_2(x)$ holds for all $x \in V_{in}$, then endochrony ensures that $s_1(clk(x)) = s_2(clk(x))$ holds for all $x \in V_{in}$. By determinism of $R_{buf}$, we then have $s_1(y) = s_2(y)$ and $s_1(clk(y)) = s_2(clk(y))$ for all $y \in V_{out}$.

However, this is also the case for all states of the state transition system of $R_{DPN}$ of component $nwe$ in Figure 7(b), and is therefore not sufficient to guarantee an isomorphism between $R_{buf}$ and $R_{DPN}$. Moreover, $nwe$ also alerts us that we should not replace the criteria of checking endochrony by checking determinism of $R_{DPN}$. Consider a sequence of inputs $input_1$ in $R_{buf}$:

$$\begin{array}{c|c|c}
\text{input}_1 & \text{input}_2 \\
\hline
x_1 & & 1 \\ 
x_2 & & 1 \\ 
x_3 & 0 & 0 \\
\end{array}$$

In the real implementation of the data-driven component of $nwe$, the sequence of input values is shown in $input_2$, since $[0]$ means that the first 0 at the head of buffer $x_3$ is not consumed in the first cycle. However, because of the nondeterminism introduced by merging $s_2, s_3$ of $R_{buf}$ in Figure 7(a), there exists also another schedule instruction, i.e. $\{clk(x_1) = false, clk(x_2) = true, clk(x_3) = true\}$. If this is the instruction the data-driven $nwe$ chose, then $\{x_2 = 1, x_3 = 0\}$ will be consumed, and after this reaction the values left at the head of input buffers are: $\{x_1 = 1, x_2 = 0, x_3 = 0\}$. However, under this situation, no guarded action is able to fire, and $nwe$ is deadlocked. By Theorem 1 instead, we know that such a case would never happen to an endochronous component, since a unique schedule instruction is determined, which is the one specified in $R_{buf}$. As an example, the endochronous component Sequential-Or is shown in Figure 9 and its transition systems in Figure 10, where $x_1, x_2$ are the input variables and $y$ the output variable.

$$\begin{array}{c}
(\text{clk}(x_1) \& \text{clk}(x_2) \& x_1) \rightarrow y = (x_1, \text{true}) \\
(\text{clk}(x_1) \& \text{clk}(x_2) \& x_1) \rightarrow y = (x_2, \text{true}) \\
\end{array}$$

Fig. 9: guarded actions of Sequential-Or.

$$\begin{array}{c}
\text{s1} & 1 & 0 & 1 \\
\text{s2} & 1 & 1 & 1 \\
\text{s3} & 0 & 0 & 0 \\
\text{s4} & 0 & 0 & 1 \\
\text{s1}' & 1 & 1 & 1 \\
\text{s2}' & 1 & 1 & 1 \\
\text{s3}' & 0 & 0 & 0 \\
\text{s4}' & 0 & 0 & 1 \\
\end{array}$$

(a) (b)

Fig. 10: Transition System of Sequential-Or.

It is clearly seen that for Sequential-Or, $R_{DPN}$ preserves all transitional information of $R_{buf}$.

Looking back, we traveled through three levels of abstractions: the first level is our starting point, which is a synchronous clock-driven component. In order to perform desynchronization, we first moved to the second level, where clocks are treated as schedule instructions. We showed that a clock-scheduled component is not really different from its clock-driven version, although it already works asynchronously. Finally we remove clocks by existential quantification to derive the third level – data-driven components – our implementation
target, and we proved that no nondeterminism is introduced only if the component is endochronous.

V. EXPERIMENTAL RESULTS

We evaluated our method of checking endochrony on a set of examples, including both endochronous and non-endochronous clock-driven components. As outlined in section IV-D, we implemented a set of algorithms to check if endochrony defined in Definition 6 holds for $R_3$. Note that the quantifier over states can be directly eliminated. Then we invoke a SAT solver to check the validity of the formula. In case it is valid, we know that the corresponding component is endochronous. In particular, we employed both a BDD based solver (NuSMV [37]) and a SAT solver (Z3 [38]) to verify our property. In order to keep a fair comparison, we boolified all non-bool programs so that a pure boolean formula of (2) can be generated, and then we push this boolean formula into NuSMV and Z3 respectively to check its validity. The whole set of experiments is carried out on a laptop with an 2.9 GHz Intel Core i7 processor and 8 GB DDR3 memory. Table I shows the preliminary result of our verification.

<table>
<thead>
<tr>
<th>Example</th>
<th>BDD (NuSMV)</th>
<th>SAT (Z3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time/sec</td>
<td>node</td>
</tr>
<tr>
<td>Seq-Or</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>Seq-ITE</td>
<td>0.01</td>
<td>1</td>
</tr>
<tr>
<td>Par-Or</td>
<td>0.01</td>
<td>268</td>
</tr>
<tr>
<td>Par-ITE</td>
<td>0.01</td>
<td>1,053</td>
</tr>
<tr>
<td>NWE</td>
<td>0.01</td>
<td>368</td>
</tr>
<tr>
<td>Gustave</td>
<td>0.02</td>
<td>1</td>
</tr>
<tr>
<td>Filter</td>
<td>0.12</td>
<td>14,645</td>
</tr>
<tr>
<td>OpDecode</td>
<td>722.05</td>
<td>1</td>
</tr>
<tr>
<td>Heating</td>
<td>9.91</td>
<td>1</td>
</tr>
</tbody>
</table>

Our test suit contains both small examples and practical applications that are written in Quartz [34]. The first four test cases are sequential and parallel versions of Or and If-Then-Else operations, where the sequential versions are endochronous. Gustave function [39] is a typical example that is endochronous but not sequential. As far as we know, none of the existing methods for checking endochrony covers Gustave function. NWE is the example we discussed before. The rest three examples are based on practical applications. Filter is a component that only chooses to read from a subset of its inputs at each cycle. Since the choice depends on absent of signals that makes it non-endochronous. OpDecode models the decoding stage of a processor. It has input values $i_1, i_2$ and $i_3$ that form a “clock-tree” where $i_3$ is the master clock and $i_2, i_3$ are its sub-clocks. In particular, if $i_1 = true$, then $i_2$ determines the rest of present values, otherwise $i_3$. Heating is a component that reads indoor and outdoor temperatures as well as movements inside a house, and based on different conditions, it either increases the indoor temperature or does nothing. Inside the table, whether or not the component is endochronous is shown in the right-most column. Performances of the two solvers are shown in the middle columns. For both solvers we record the time (in seconds) taken to compute the final result. For NuSMV, we record the size of the BDDs generated, and for Z3, we record the number of clauses made. It is noticeable that once a formula is valid, its BDD reduces to a single node which is true. However, for some endochronous example it still takes a long time to compute this single node (e.g. OpDecode). On the other hand, the SAT solver Z3 performs quite well in all cases. In order to test scalability of our method, we further generated a set of parameterized test suit. The result is shown in Table II.

<table>
<thead>
<tr>
<th>Param.</th>
<th>No. vars</th>
<th>BDD (NuSMV)</th>
<th>SAT (Z3)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time/sec</td>
<td>nodes</td>
<td>clause</td>
</tr>
<tr>
<td>(3,2)</td>
<td>50</td>
<td>0.01</td>
<td>268</td>
</tr>
<tr>
<td>(4,4)</td>
<td>150</td>
<td>282.53</td>
<td>1,108,464</td>
</tr>
<tr>
<td>(4,8)</td>
<td>214</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(4,32)</td>
<td>342</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(8,32)</td>
<td>634</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(16,32)</td>
<td>1,218</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(32,32)</td>
<td>2,386</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(64,32)</td>
<td>4,772</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(128,32)</td>
<td>9,394</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(256,32)</td>
<td>18,738</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>(256,64)</td>
<td>22,330</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

In particular, we chose Parallel-Or as our template and two parameters $(m, n)$, as shown in the first column. The first parameter $m$ equals the number of rules (which equals the number of guarded actions) that are generated as well as the number of input variables, and the second parameter $n$ bounds the variables by the range of natural numbers $[0, n - 1]$. Each generated rule only choose to read one of the $m$ input variables and assigns it to the single output. Also for each program, there is an additional rule specifying if all inputs are present and 0, then 0 is the output. Therefore, all parameterized programs are not endochronous. Besides data collected similar to Table I, we also record the number of free variables in formulas in the second column. The data in Table II shows that our method scales quite well for SAT as it can deal with programs having 20,000 boolean variables within three seconds, while the BDD solver soon become unavailable for cases with 200 variables. In particular, from parameters $(4, 8)$, we run our tests on NuSMV for 10 minutes and terminate without getting any result.

VI. CONCLUSION AND FUTURE WORKS

In this paper, we have developed an efficient decision procedure to verify whether a given synchronous multi-clocked component is endochronous, i.e., whether we can run the synchronous code generated for that component in an asynchronous setting that is completely driven by the arrival of input data. For the correctness proofs, we considered the component at different levels of abstraction, one where inputs are buffered and a second where even the notion of clocks has been removed. We proved that the component is endochronous if and only if the removal of clocks does not introduce nondeterminism.

In our future work of the paper, we want to generalize the results of this paper to consider weak endochrony. In general, weakly endochronous components will introduce nondeterminism after the removal of the clocks, but only nondeterminism in terms of trace theory, i.e., different scheduling...
of independent actions. After these checks, it is then possible to generate multi-threaded code – one thread per component – without destroying the originally verified behaviors of the synchronous models.

**References**


