Monitoring Distributed Reactive Systems

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

Abstract—Recent results on the desynchronization of synchronous systems introduced the subclass of so-called endo/isochronous systems. Since the modules of these systems can derive their own local clocks from their inputs, they can be implemented as asynchronous components without inefficient synchronizations. Runtime verification has a similar problem since one has to add a monitor to an existing system where typically a synchronous composition is assumed.

In this paper, we therefore propose to use the endo/isochronous system theory to implement asynchronous monitors that are still able to check the original synchronous behavior. We only demand that the entire system obtained by adding the monitor is an endo/isochronous system so that we can implement a suitable wrapper around the monitor (and the other components). For the implementation, we use heuristics to check endochrony like the sequentiality of the implemented functions.

I. INTRODUCTION

Compared to traditional software design, the design of embedded systems is even more challenging: In addition to the correct implementation of the systems, one has to consider non-functional constraints like real-time behavior, reliability, and energy consumption. For this reason, embedded systems are often implemented with heterogeneous processors to optimize the mentioned criteria. This has the consequence that many embedded systems become parallel systems. Moreover, to avoid expensive synchronizations between the components, these parallel systems run asynchronously to each other, and they are therefore distributed systems (even though they may still be contained in a small system).

Moreover, many embedded systems are used in safety-critical applications where errors can lead to enormous damages and even to the loss of human life. For this reason, formal verification is applied in many design flows using different kinds of formal verification methods. Most of these formal methods assume a synchronous composition of parallel modules like most kinds of model checking temporal logics including LTL, CTL and most others [12].

For this reason, it is natural to develop model-based design flows for embedded systems where one starts with a synchronous model. Based on that synchronous model, one can perform formal verification by directly applying known verification procedures. For the further implementation, one then has to desynchronize the synchronous system so that it can be implemented by asynchronous components.

Recent results on desynchronization lead to the definition of endo/isochronous systems [5–8, 27–29] of Benveniste et al. The main idea is thereby that a synchronous component can derive its own local clock by observing the presence of certain input patterns. Once a legal input pattern is seen, the corresponding input values are consumed and a synchronous reaction of the component can take place. Provided that a synchronous composition of modules is endo/isochronous, one can easily implement it as an asynchronous and distributed system. To this end, one simply has to add a wrapper around each component that is essentially responsible for generating its local clock from the presence of certain inputs. The local clock tick triggers then the already given synchronous behavior inside the component.

While design flows that generate asynchronous implementations from synchronous models by desynchronization have already been proposed (but not really implemented), the theory is still new and its applications have not yet been fully explored. In this paper, we consider a further application of this theory, which leads to a more efficient way of runtime verification of distributed reactive systems.

Runtime verification [1, 18–22] is often a good compromise between formal verification and testing. In runtime verification, one adds special components called monitors to a system whose task is to check whether required properties are satisfied during the execution of the system. For most standard temporal logics (at least for safety properties), a property \( \varphi \) can be automatically translated to an equivalent monitor [14, 19, 20, 26]. The resulting component can then either check the system online, i.e. the monitor and the system run in parallel (the typical case), or offline, where the monitor checks a finite set of previously recorded executions.

A particular problem for runtime verification is that the decisions of monitors can only be made on the finite history of the executions seen so far, while the semantics of temporal logic formulas requires to consider the unknown infinite future as well. For this reason, many special temporal logics have been developed [2, 4, 25, 26]. These logics extend the boolean values true and false by further logic values that denote some further information about the unknown future. All of these logics still assume a synchronous composition with the monitors.

By definition of the semantics of temporal logics, the monitors run synchronously in parallel to the monitored system. For this reason, one is forced to synchronize the monitor and the monitored system so that both subsystems influence each other. Since the monitor often checks the consistency with a less efficient but more readable specification, this typically slows down the overall performance and therefore worsens non-functional properties of the system. The approach presented in this paper removes the synchronization between the system and the monitor even though the temporal logics used for specification are still based on a synchronous semantics.

In this paper, we therefore propose an approach that we call separate monitoring. It adds monitors for runtime veri-
fication of reactive systems that were originally based on a synchronous composition, but are then implemented as asynchronous/distributed systems. We create separate components for the monitors that run asynchronously to the remaining components (that might also run asynchronously to each other). The components of the system are not modified at all so that the real-time behavior of the system and most other non-functional properties are preserved. In real systems, e.g., in an automotive application, the monitor can be implemented on an additional control unit, snooping the vehicle bus for runtime-verification. Obviously, to cope with asynchronous communication, we need to ensure additional system properties so that the monitor always has a consistent view of the system state.

To summarize, the main contribution of this paper is a methodology for separate monitoring of distributed reactive systems that were obtained by desynchronization of an original synchronous composition of modules that now run asynchronously to each other. Based on the theory of endo/isochronous systems, we define the subclass of synchronous systems that can be separately monitored, and we show how to construct an asynchronous monitor component for a given property. Surprisingly, it turns out (see end of Section III) that even systems can be separately monitored that are not endo/isochronous.

The rest of this paper is organized as follows: In Section II, we list all definitions required to explain the theory of endo/isochrony. Even though this section contains several modifications and clarifications of the original definitions, we do not consider this section as an added value per se. Section III presents the main part of the paper, i.e., how to implement separate monitoring of distributed reactive systems. In Section IV, we conclude with a short summary.

II. FOUNDATIONS

In this section, we reconsider previous work on the theory of endo/isochronous systems. In Section II-A, we first describe the basic principles of globally asynchronous locally synchronous (GALS) systems and introduce the notation we use in this paper. Section II-B reviews the main results of the theory of endo/isochronous systems which are used to desynchronize synchronous systems for asynchronous/distributed implementations. Section II-C sketches the foundations of our approach to monitoring distributed systems.

A. GALS Systems

In this paper, we aim at monitoring systems that follow the globally asynchronous locally synchronous (GALS) [11, 15] principle. As the name suggests, such systems consist of components that follow internally the synchronous model of computation [9, 17], but interact with each other in an asynchronous manner. The execution of a synchronous system consists of a sequence of reactions \( \mathcal{R} = \langle R_0, R_1, \ldots \rangle \) where all inputs are read and all outputs are immediately written by the synchronous system. Thus, it basically implements a function mapping input values to output values depending on some local state. In contrast, communication between distributed components is asynchronous: there is no shared memory, and information is only exchanged with the help of FIFO buffers. Thereby, the reactions of the individual components are decoupled, each one triggering its own computation according to the availability of suitable input values.

GALS systems can be seen as a special case of dataflow process networks (DPN), which we will use in the rest of this paper to describe our examples. The behavior of a DPN is usually described with the help of firing rules (see the example in Figure 1), which are a simple operational description of a single component. In each step, the component basically matches the current content of the input buffers to the patterns given in the left half of the table. Among the rows with matching input patterns, an arbitrary one is selected, the matched input values are removed from the input buffers, and output values are written to the output buffers according to the right half of this row. Throughout the paper, \([a]\) denotes a list containing the element \(a\), \([a,b,c]\) denotes a list containing the elements \(a, b,\) and \(c\), and finally, \(a :: A\) denotes a list starting with element \(a\) and the remaining list \(A\).

In the case of GALS systems, the patterns on the left-hand side and the output list always contain at most one token, since fully synchronous components are only allowed to read and write at most one value for each signal in each reaction. For technical reasons, we add to the set of data values \(D\) a special symbol \(\Box\), which is used if no value is read or written in a particular step. This will simplify the presentation in the following since we achieve this way that always exactly one value is consumed and produced for each input and output signal even though the realized system may not read or write the values denoted as \(\Box\). Obviously, this special value \(\Box\) will be never transmitted in the implementation of the GALS system: matching \(\Box\) on the left-hand side means to read nothing from the corresponding input channel, and a \(\Box\) in the output list means that no value is produced for the corresponding output channel.

Consider the example shown in Figure 1, which consists of three components even, cpy and add. The component even produces a stream of all even numbers (provided that \(z_1\) initially contains the token with value \(0\)), cpy simply copies values from the input buffer to the output buffer and add adds the inputs when both inputs are present, or it forwards the present value to the output. The firing rules given on the right-hand side of the figure describe this behavior. The node even takes a token with value \(i\) from the input buffer (as described by the pattern \(i :: A\)), and puts a token with value \(i+2\) to both output buffers. An example trace of the system is therefore the following one:

| \(z_1\) | 0 | 2 | 4 | 6 | 8 | 10 | ... |
| \(z_2\) | 1 | \(\Box\) | 2 | 3 | \(\Box\) | 2 | ... |
| \(z_3\) | 0 | 2 | 4 | 6 | 8 | 10 | ... |
| \(z_4\) | 1 | \(\Box\) | 2 | 3 | \(\Box\) | 2 | ... |
| \(y\) | 1 | 2 | 6 | 9 | 8 | 12 | ... |

B. Endochrony and Isochrony

Encoding and transmitting \(\Box\) is not desired since it involves a large overhead for implementations. Instead, as already mentioned above, these tokens representing the absence of values are mapped to absence of transmission, i.e. we do not want to send and receive the \(\Box\) tokens. However, in the following formalizations, we keep them to study the effect of delays on the asynchronous communication channels. Obviously, as these \(\Box\) values align the streams in a particular way, it is an interesting question, which systems still show the same
behavior if we introduce additional \( \square \) values or remove them completely.

This fundamental problem of desynchronization has been addressed in previous work [5–8, 27–29], and the notions of endochrony and isochrony are intended to solve this problem. Before we continue with a discussion of these concepts, we first make some auxiliary definition over streams, which we need to define these concepts.

1) **Stretch and Flow Equivalence:** In a GALS systems, local components generally run at different speeds, and some components might have to wait for inputs to perform a reaction step. With the help of the \( \square \) tokens, this can be modeled by so-called silent reactions, i.e., reaction steps produced by firing rules where each pattern has the form \( \square :: L \) and each output is \( \square \). Thus, when a silent firing rule is fired, neither inputs are consumed nor outputs are produced. As these silent reactions do not contribute to the real behavior, we often want to omit them.

Let \( x = \text{Bhv}(P) \) be a behavior of process \( P \), i.e., \( x \) is a tuple of streams \( x \in \text{Stream}^m \) for all variables in \( P \). Then, the behavior \( \text{stfree}(x) \) is the one where all silent reactions of \( s \) have been removed. Furthermore, we say that \( x \) and \( y \) (both in \( \text{Stream}^m \)) are stretch-equivalent \( (x =_{st} y) \) if and only if \( \text{stfree}(x) = \text{stfree}(y) \). For example, \((x_1, x_2, x_3) =_{st} (y_1, y_2, y_3)\):

\[
\begin{array}{c|c|c|c|c|c}
  x_1 & 0 & 2 & 6 & 10 & \ldots \\
  x_2 & 1 & \square & 3 & 2 & \ldots \\
  x_3 & 0 & 2 & \square & 8 & 7 & \ldots \\
  y_1 & 0 & \square & 2 & 6 & 10 & \ldots \\
  y_2 & 1 & \square & 3 & 2 & \ldots \\
  y_3 & 0 & \square & 2 & 8 & 7 & \ldots \\
\end{array}
\]

As already stated above, we want that components of the GALS system only consume and produce real values (of domain \( D \)), and that \( \square \) is never transmitted. To this end, we define for a stream \( x \in \text{Stream}^m \) another stream \( \text{flows}(x) \) where all \( \square \) have been removed from \( x \) (i.e., the stream that is visible in the actual implementation). Furthermore, we say that \( x \) and \( y \) are flow-equivalent \( (x =_{fl} y) \) if and only if \( \text{flows}(x) = \text{flows}(y) \). For example, \( \text{flows}(x_1, x_2, x_3) = \text{flows}(y_1, y_2, y_3) = (z_1, z_2, z_3) \) is:

\[
\begin{array}{c|c|c|c}
  z_1 & 0 & 2 & 6 \\
  z_2 & 1 & 3 & 2 \\
  z_3 & 0 & 2 & 8 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c}
  z_1' & i & i+2 & i+2 \\
  z_2' & i+2 & i+2 & i+2 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
  z_2 & z_4 & y \\
  (a :: A) & \text{cpy}(z_2) & a \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c}
  x_1 & x_2 & x_3 & y \\
  (1 :: A) & (0 :: B) & (\square :: C) & 1 \\
  (\square :: A) & (1 :: B) & (0 :: C) & 1 \\
  (0 :: A) & (\square :: B) & (1 :: C) & 1 \\
  (0 :: A) & (0 :: B) & (0 :: C) & 1 \\
 (1 :: A) & (1 :: B) & (1 :: C) & 1 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
  x_1 & x_2 & x_3 & y_1 & y_2 & y_3 \\
  (1 :: A) & (0 :: B) & (\square :: C) & 1 & 1 & 1 \\
  (\square :: A) & (1 :: B) & (0 :: C) & 1 & 1 & 1 \\
  (0 :: A) & (\square :: B) & (1 :: C) & 1 & 1 & 1 \\
  (0 :: A) & (0 :: B) & (0 :: C) & 1 & 1 & 1 \\
 (1 :: A) & (1 :: B) & (1 :: C) & 1 & 1 & 1 \\
\end{array}
\]

Fig. 1. DPN: structure and local firing rules

As a consequence, stretch equivalence implies flow equivalence, i.e., \( x =_{st} y \rightarrow x =_{fl} y \), whereas the other direction does not hold in general.

2) **Endochrony:** Now assume that the components work synchronously (i.e., still use the \( \square \) tokens) and that the communication is asynchronous (i.e., the \( \square \) values are not transmitted). In order to perform a synchronous reaction step correctly, each component must be able to reconstruct the inputs of that step from the asynchronous communication, i.e., it must be able to reinsert the deleted \( \square \) values. Endochronous components have this property: they reconstruct reaction steps (up to stretch-equivalence) in a unique way from the asynchronous input streams.

Following the formal definition of [31], we say that a synchronous component \( P \) is endochronous if and only if for any behaviors \( \rho_1, \rho_2 \in \text{Bhv}(P) \)

\[
\rho_1 =_{fl} \rho_2 \Rightarrow \rho_1 =_{st} \rho_2
\]

where \( \rho_1 =_{fl} \rho_2 \) is an abbreviation of \( \rho_1|_{V_{in}} =_{fl} \rho_2|_{V_{in}} \) and \( \rho_1|_{V_{in}} \) is the projection of \( \rho \) to the input streams \( V_{in} \) of \( P \). In words, the component \( P \) is endochronous if two flow-equivalent inputs lead to stretch-equivalent behaviors of \( P \).

In order to understand this definition, consider a variant of Berry’s Gustave function as given in Figure 2. This component is endochronous, i.e., flow-equivalence of inputs implies the stretch-equivalence of process behaviors. It reconstructs the alignment of streams by looking at all present values at the inputs (without removing them), and then chooses the appropriate firing rule. For example, if we have the input

Fig. 2. Example: Gustave
streams \((1 :: A), (0 :: B), (0 :: C)\), the process glimpses all the three input streams to finally see that the first firing rule will be chosen. For the next synchronous reaction, the component takes the 1 for \(x_3\), the 0 for \(x_0\), and introduces a \(\square\) for \(x_3\).

If the component is represented by firing rules, as in our case, it is simple to check its endochrony. For any pair of rules, we must check whether they do not overlap, i.e. if \(v_1 \neq \square\) and \(v_2 \neq \square\). Intuitively, this input can then be used to distinguish the situation where either rule is fired. The requirement that \(v_1\) and \(v_2\) are not \(\square\) ensures that we do not fire different rules depending on the delays in the streams; two flow equivalent input behaviors should lead to the same result.

Endochrony is not compositional, i.e. if we merge two endochronous components, the resulting component may not be endochronous. To see this, consider the trivial example cpy1, which only forwards data values.

\[
\begin{array}{|c|c|c|}
\hline
x & y & \{a :: A\} \\
\hline
1 & 2 & 3 \square \\
\hline
y & 1 & 2 \square \\
\hline
x & 8 & 4 \square \\
\hline
\end{array}
\]

Obviously, cpy1 is endochronous. Now, we merge two of these components to a single component cpy2:

\[
\begin{align*}
y_1 & = \text{cpy1}(x_1) \\
y_2 & = \text{cpy1}(x_2)
\end{align*}
\]

This composition destroys endochrony. We see that for cpy2 it is now the case that \(\rho_1 = \rho_2\) holds, but we do not have \(\rho_1 = \rho_2\).

\[
\begin{array}{|c|c|c|c|c|c|c|}
\hline
\rho_1 & x & y & (a :: A) \\
\hline
& 1 & 2 & 3 & \square & 5 & \ldots \\
\hline
& 1 & 2 & 3 & \square & 5 & \ldots \\
\hline
& 8 & 4 & 2 & \square & 3 & \ldots \\
\hline
& 8 & 4 & 2 & \square & 3 & \ldots \\
\hline
\end{array}
\]

As a consequence, we are not able to establish a unique synchronous execution for several endochronous components.

3) Isochrony: As we are generally not able to reconstruct a synchronous reaction for several endochronous components, we need another useful criterion in the context of desynchronization. Essentially, a desynchronized system only needs to ensure that the reconstructed synchronization for a component complies to one of the other components in the network. If several components agree on their reconstructions, they are said to be isochronous. Formally, several components \(P_1, \ldots, P_n\) are isochronous [31] if

\[
\text{flows(Bhv}(P_1 \parallel \cdots \parallel P_n)) = \text{flows(Bhv}(P_1)) \parallel \cdots \parallel \text{flows(Bhv}(P_n))
\]

Note that the composition of flows on the right-hand side is an asynchronous composition, but semantically it is the same as the synchronous composition (as the only difference is that the streams that are composed are already free from \(\exists\) values). Isochronous compositions guarantee that the asynchronous composition generates a behavior that is flow-equivalent to the one of the synchronous composition. Note that isochrony is not composable, i.e. composing two sets of isochronous components is not necessarily isochronous. This is obvious since a single component is always isochronous by definition so that compositionality would make all systems isochronous.

In example cpy2 (see Section II-B), the composition of both cpy1 components is apparently isochronous. Checking the property in general can be done by model checking [7, 31]. For all reactions of the endochronous components \(P_1, \ldots, P_n\), we must check that, whenever they agree on the present shared variables (shared variables that are different to \(\square\) in the current reaction), then they also agree on all the shared variables. This guarantees that the receiver inserts \(\square\) tokens at exactly the same points of the stream where the sender at the other end of the asynchronous communication channels has discarded them.

C. Monitoring

As already outlined in the introduction, there is a lot of related work on runtime verification, and our approach is based on many previous results. In order to make the paper self-contained, we briefly review some previous results, which will be used in the rest of the paper.

In general, a monitor reads the inputs and outputs of a monitored system, i.e. it is given a trace of the system. Many different formalisms [14] have been used to specify the properties over traces, which are checked by monitors, e.g. LTL [3], ptLTL [23, 24], extended regular expressions [13] and many others. In the following, we focus on ptLTL, both due to its wide use in runtime verification and for its simplicity (which avoids distraction so that the focus can be kept on our actual contribution, embedding monitors into an asynchronous context).

The syntax of ptLTL is defined as follows. Any atomic proposition is a ptLTL formula. If \(\phi\) and \(\psi\) are ptLTL formulas, then \(\overline{\phi}\) (previously), \(\overline{\phi} \land \overline{\psi}\) (eventually in the past), \(\overline{\phi} \land \overline{\psi}\) (always in the past) and \(\overline{\phi} \land \overline{\psi}\) (before) are also ptLTL formulas. The fact that a ptLTL formula \(\phi\) holds for a given trace \(t\) is denoted by \(t \models \phi\), where \(t = s_1, s_2, \ldots, s_n\) a finite sequence of states, which is formally defined as follows:

- \(t \models \text{true}\) is always true
- \(t \models \text{false}\) is always false
- \(t \models \alpha\) where \(\alpha\) is a proposition if \(\alpha\) holds in state \(s_n\)
- \(t \models \overline{\alpha}\) if \(t \not\models \alpha\)
- \(t \models \phi \land \psi\) if \(t \models \phi\) and \(t \models \psi\)
- \(t \models \phi \lor \psi\) if \(t \models \phi\) or \(t \models \psi\)
- \(t \not\models \overline{\phi}\)
- \(t \not\models \overline{\phi}\)
- \(t \models \overline{\phi}\) if \(t \not\models \phi\)
- \(t \models \overline{\phi}\) if \(t \models \phi\) and \(t \models \psi\)
- \(t \not\models \overline{\phi}\)
- \(t \models \overline{\phi}\) if \(t \not\models \phi\)
track of its current and previous value, which allows it to directly implement the semantics of \( \text{pTLTL} \) operators.

The example also shows that the synthesized monitor core reads the trace of the system step by step, i.e., values are consumed reaction by reaction. Hence, the most difficult part of centralized monitoring for distributed systems is the creation of a monitor wrapper, which is responsible for the reconstruction of the reactions lost in the asynchronous communication. Once communication is resynchronized, the reactions can be forwarded to the monitor core, so that it processes the next input set in its next iteration.

### III. MONITORS FOR DISTRIBUTED REACTIVE SYSTEMS

In this section, we show how the results from the previous section can be used to realize the separate monitoring, which we propose in this paper.

#### A. General Idea

Section II-C explained how monitors can be automatically generated from a given temporal logic formula (see Figure 3). As we use the same logic in our approach, we use exactly the same techniques to generate the core of our monitors. However, our monitors additionally need a wrapper around the generated core which converts the asynchronous streams to synchronous reactions. This wrapper is obviously necessary since new input values for the monitor do not arrive reaction by reaction. Instead, the asynchronous communication delays variables differently in general, and \( \square \) tokens are even never transmitted.

As the generated core of the monitor depends on reading its variables reaction by reaction, the wrapper must resynchronize incoming events from the asynchronous FIFO buffers, i.e., it waits until all events of a reaction arrive, update the variables of the monitor core and then triggers its next iteration (if the wrapper reconstructs a \( \square \) for a variable, the corresponding variable in the monitor core remains unchanged). Obviously, the crucial part of the wrapper is to find out when all events of a reaction have arrived since there are no explicit events for \( \square \) values in a reaction. Thus, it must be able to distinguish between a non-existing event (due to \( \square \)) for the reaction and a delayed event (due to communication delays).

This is the point where the theory of endochrony and isochrony (see Section II-B) comes into play. If the monitor fulfills certain criteria, it can accomplish the task of resynchronization, which is the crucial part of separate monitoring.

The first requirement for our monitor is endochrony. It ensures that we can extract stretch-equivalent reactions from flow-equivalent inputs, i.e., that different delays of inputs (flow-equivalence) still allow us to reconstruct reactions in a unique way (modulo stuttering, i.e., silent actions). Thus, we can use traditional monitors as our core, and we only need to add an appropriate wrapper.

The second requirement for our monitor is that its composition with the system is isochronous, i.e., the asynchronous composition returns a flow-equivalent result as the synchronous composition. As a consequence, the system behavior which is observed by the monitor corresponds to the actual one, and the unique way the endochronous wrapper reconstructs reactions complies with the reactions of the system.

Adding a wrapper that makes the monitor endochronous, and ensuring that its composition with the system is isochronous, guarantees that we get the same verification result as if the monitor would have been synchronously composed to the system (we only lose the \( \bar{X} \cdot \) due to the potential stuttering of the reconstructed reactions). Thus, the correctness of our approach is a direct consequence of the theory of endochrony and isochrony.

#### B. Constructing Wrappers for Monitors

In this section, we show how a wrapper satisfying the requirements mentioned above can be constructed. For the sake of simplicity, first assume that we want to monitor a single (endochronous) component of a GALS system. We must rearrange its inputs and its outputs to reactions (i.e., the monitor is endochronous), and the newly created reactions of the monitor must be the stretch-equivalent to the original ones of the component (i.e., the composition of component and monitor is isochronous).

For the construction of the wrapper, we can follow the strategy presented in [10, 16] for the distribution of synchronous programs. In order to resynchronize the inputs of the monitored endochronous component, we can simply reuse the wrapper of the component itself. Then, the wrapper of the monitor will reconstruct exactly the same reactions by construction. However, the wrapper also needs to reconstruct the alignment of the outputs. For this part, we generally need the part of the component that is responsible for setting the clock of the outputs: if we add the cone of influence for the output clocks to the monitor wrapper, we also get the same reactions by construction.

Apparently, this solution means in the worst case that we need to integrate all the computations of the monitored component in the wrapper of the monitor, which is however a rather pathological case (and even in this case, it is not a significant problem for our approach, since the additional computational effort is still limited to the monitor, which is decoupled from the observed system).

Alternatively, we can use the firing rules of the monitored component to construct a wrapper for the inputs and outputs. Figure 4 illustrates the general construction. The wrapper continuously keeps track of the first values of all monitor inputs (realized by first). Naturally, buffers may be empty, in which case \( \square \) is returned. Then, the wrapper matches these values with the patterns of the firing rules of the monitored component (see Figure 2), which is implemented by the if

\[
\begin{array}{l}
(p, q) := \text{GetNextReaction}(); \\
\text{cur}_3 := (z_1 = 1); // subformula \varphi_3 : (z_1 = 1) \\
\text{cur}_2 := (s_0 > 15); // subformula \varphi_2 : (s_0 > 15) \\
\text{cur}_1 := \text{cur}_3; // subformula \varphi_1 : \mathcal{F} (s_0 > 15) \\
\text{output} (\text{cur}_3 \rightarrow \text{cur}_1); \\
\text{loop} (p, q) := \text{GetNextReaction}(); \\
\text{pre}_3 := \text{cur}_3; \text{cur}_3 := (z_1 = 1); \\
\text{pre}_2 := \text{cur}_2; \text{cur}_2 := (s_0 > 15); \\
\text{pre}_1 := \text{cur}_1; \text{cur}_1 := \text{pre}_1 \lor \text{cur}_2; \\
\text{output} (\text{cur}_3 \rightarrow \text{cur}_1);
\end{array}
\]

Fig. 3. Monitoring \( \varphi_0 \)
statements (as our wrapper is endochronous, we never match with $\Box$). The matched inputs are removed from the input buffers and the corresponding variables of the monitor core are updated (both realized by read). Finally, the core is triggered, which can now check the next reaction.

Next, we remove our initial assumption that we only want to monitor a single endochronous component. Thus, we now have several endochronous components, which are isochronous to each other. In this case, exactly the same strategy can be used. Instead of matching with the firing rules of a single component, we match with the firing rules of all components. If any of these matches succeeds, the monitor core is triggered. For more details about the implementation of the wrappers, see Section III-D below. Needless to say that also the other strategies from [10, 16] can be applied to this general situation.

C. Endochrony of Monitors

Each monitor $M_\phi$ is responsible for verifying a given property $\phi$. For this task, its wrapper needs to resynchronize all the variables appearing in $\phi$ (let us call them $V_\phi$) before they are handed over to the core for the actual monitoring. Thus, according to the definition of endochrony from Section II-B2, the monitor $M_\phi$ is endochronous if and only if flow equivalent behaviors $\rho_1, \rho_2$ of $V_\phi$ lead to stretch equivalent resynchronizations: $\rho_1 = \rho_1^{\top \top}, \rho_2 = \rho_2^{\top \top}$.

Since the inputs of the monitors and the monitored system are generally different, an endochronous system does not imply an endochronous monitor or vice versa (in principle).

Let us first have a closer look on one direction: a monitored system may be endochronous while its monitor is not. For example, consider the following endochronous system:

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$y_1$</th>
<th>$y_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \cdot A$</td>
<td>$1 \cdot B$</td>
<td>$a \neq \Box$</td>
<td>$a$</td>
</tr>
<tr>
<td>$a \cdot A$</td>
<td>$0 \cdot B$</td>
<td>$a \neq \Box$</td>
<td>$\Box$</td>
</tr>
</tbody>
</table>

A monitor only reading the variables $V_\phi = \{ x_1, y_1, y_2 \}$ will not be endochronous since it is impossible to reconstruct the reactions without $x_2$. Obviously, this effect can avoided by giving the monitor the missing information, i.e. adding the missing variables to its interface ($x_2$ in our case). These variables are only used for the reconstruction of the reactions in the wrapper but they are never forwarded to the wrapper. Thus, endochronous systems can be generally separately monitored since the monitor can be always given enough information for the resynchronization.

On the other hand, we can monitor systems that are not endochronous. This is due to the fact that the monitor can also use the system outputs (which are included in its inputs) for the resynchronization. For example, consider the following:

\[
\begin{array}{cccc|cc|c}
\phi_0 & x_1 & x_2 & y_1 & y_2 \\
(a \cdot A) & (1 \cdot B) & a \neq \Box & a & [1] \\
(a \cdot A) & (0 \cdot B) & a \neq \Box & \Box & [0] \\
(a \cdot A) & (\Box \cdot B) & a \neq \Box & \Box & [2] \\
\end{array}
\]

Apparently this example is not endochronous because the third firing rule is not orthogonal to the previous ones. Since the absence of $y$ cannot be distinguished from its delay, a premature firing of the third rule may be possible. However, when taking the outputs into account, the behavior of the example can be resynchronized: depending on the value of $y_2$, we can distinguish all three firing rules. Hence, this example can be monitored without problems: with the help of $y_2$ (and $x_2$), the monitor is endochronous and the composition is isochronous by construction.

D. Examples

The first example is the system Sampler − Alert shown in the upper part of Figure 5. It is obviously endochronous: in every step, $s_0$ is sampled by $c_0$ (Sampler0), and the resulting $y_0$ is sampled again by the subsequent Alert.

As the system is endochronous, the wrapper of the monitor can reconstruct the synchronous reactions as shown in the lower part of Figure 5. From the values arriving over the asynchronous buffers ($\rho$) the synchronous executions of Sampler and Alert ($\rho'$) are built and aligned according to the shared variables.

With the help of $\rho'$, it is straightforward for the monitor core to check any given property. For example, the monitor can check $\phi_0$, which specifies that an alert ($z_1 = 1$) is always preceded by a primary input $s_0$ greater than 15:

\[
\phi_1 : (z_1 = 1) \rightarrow \overline{F}(s_0 > 15)
\]
\[
\left\{
\begin{array}{l}
(s_0, c_1) = \text{add}(a_0, b_0, 0, m) \\
(s_1, c_2) = \text{add}(a_1, b_1, c_1, m)
\end{array}
\right.
\]

The core of this monitor is shown in Figure 3.

As a second example, consider the system shown in Figure 6. It consists of two add components that can be used separately to add two pairs of numbers with a single digit each or coupled to add numbers with two digits. The mode is controlled by the boolean flag \(m\), and the numbers are given by \(a_0\), \(a_1\), \(b_0\) and \(b_1\). The system is endochronous, and the following property (which ensures that the addition of two digit numbers is always correct) can be monitored:

\[
\phi_2 : G m \rightarrow (a_0 + b_0 + 10*(a_1 + b_1) = s_0 + 10*s_1 + 100*c_2)
\]

The core of the monitor can be obtained by automatic synthesis. For the wrapper, we need to resynchronize all the variables of the two \texttt{add} components. This can be accomplished by the wrapper shown in Figure 7. It was automatically derived from the firing rules for \texttt{add} (see Figure 6). With its help, all the variables needed for the evaluation of \(\phi_2\) can be arranged in reactions.

E. Synchronous Model vs. GALS Implementation

In the previous section, we showed how separate monitors can be constructed for distributed reactive systems. As already mentioned there, the key to resynchronize the variables of the monitored system is the endochrony of the monitor and the isochrony of its composition. In this section, we have another look at these properties and point out subtle differences between the abstraction layers of synchronous monitor model and the GALS implementation.

To start the discussion, consider once again the example \texttt{cpy2}:

\[
\left\{
\begin{array}{l}
y_1 = \text{cpy1}(x_1) \\
y_2 = \text{cpy1}(x_2)
\end{array}
\right.
\]

As already stated in Section II-B, module \texttt{cpy2} is not endochronous. Nevertheless, both submodules \texttt{cpy1} are endochronous and their composition is isochronous. Thus, it is a perfect system for a GALS implementation.

Now assume that we want to monitor the following property:

\[
\phi : G [(x_1 = x_2) \rightarrow (y_1 = y_2)]
\]

It is now interesting to see that the synchronous composition of \texttt{cpy2} and the monitor is again endochronous. The code for checking \(\phi\) implicitly aligns \(x_1\) and \(x_2\) (as well as \(y_1\) and \(y_2\)) so that all signals are synchronous. Thus, we are in a situation where the addition of a monitor changes the endo/isochrony of the entire system.

Note that a monitored \texttt{cpy2} system itself can still run completely asynchronously, the executions of two \texttt{cpy1} components are independent. Nevertheless, the monitor can reconstruct the synchronous view to the system, which is then used for runtime verification. Thus, the efficiency of GALS implementation is preserved in the presented approach, even if monitors resynchronize executions.

The function \texttt{WrapSumMonitor()} is defined as follows:

\[
\text{Function } \text{WrapSumMonitor()}
\begin{array}{l}
\text{loop }
\end{array}
\]

IV. SUMMARY

In this paper, we proposed separate monitoring for distributed embedded systems. In contrast to previous approaches, separate monitoring does neither change the system components nor its communication, which avoids that monitors slow down the performance of a system. We achieve this by completely decoupling monitors from the actual system, only communicating over asynchronous communication channels.

We explained how results on the desynchronization of synchronous systems can be used to resynchronize asynchronous behaviors so that a monitor built from a temporal logic specification can check them. This is a key issue since the monitor cores are based on synchronous reactions, which were lost in the asynchronous communication. However, as long as the monitor is endochronous, and its composition with the system is isochronous, the theory of endo-/isochrony guarantees that the synchronous view of the system can be rebuilt.

For a given property, the corresponding monitor can be automatically generated. We use the classic algorithms for the synthesis of the monitor cores, while the wrapper can be inferred from the description of the monitored system. Thereby, our methodology ensures endo-/isochrony of the generated monitors.

\[
\begin{array}{c|c|c|c|c|c|c}
 & x_1 & x_2 & x_3 & x_4 & y_1 & y_2 \\
(a : A) & (b : B) & 0 & 0 & a, b \neq 0 & [a + b] & \square \\
(a : A) & (b : B) & (c : C) & 1 & a, b, c \neq 0 & [a + b + c] & [(a + b + c) > 10]
\end{array}
\]

\text{Fig. 6. Example Sum [30]}

\text{Fig. 7. Wrapper for monitoring Sum (Figure 6)}
REFERENCES


