Rethinking of I/O-automata composition

Sarah Chabane
LIMOSE Laboratory,
Université M'hamed Bougara
Boumerdes, Algeria
chabane.sarah@univ-boumerdes.dz

Rabea Ameur-Boulifa
LTCI, Télécom ParisTech, Université Paris-Saclay
rabea.ameur-boulifa@telecom-paristech.fr

Mohamed Mezghiche
LIMOSE Laboratory,
Université M'hamed Bougara
Boumerdes, Algeria
mohamed.mezghiche@yahoo.fr

Abstract

The necessity of handling the increasing complexity of embedded systems has led to the usage of reuse-based design. At the same time, the systems must still satisfy strict requirements on reliability and correctness. This paper proposes a formal analysis of parallel composition of I/O automata. This analysis leads to identification of novel composition rules guaranteeing the correctness-by-construction, and will provide a basis for a sound compositional development of components (Intellectual Property blocks).

I. Introduction

One approach to handle the increasing complexity of circuits and systems is the reuse of existing component models. Specifying systems in a compositional manner is an established approach to cope with complexity. Additionally, formally specifying designs and using formal techniques for verification, such as the model checking technique, is an established approach to ensure high quality of design. Based on component-based approach, systems are designed as the composition of interconnected, inherently parallel component models. The composition technique we consider in this paper is cascade and synchronous composition; two or more components connected in series and reacting simultaneously.

Building computational models that naturally support compositionality is the subject of intensive studies [7], [2], [1]. One successfully promoted direction for modelling synchronous systems is the input/output automaton (I/O automaton for short) proposed by [10]. Unfortunately, I/O models semantic have a compositionality defect wrt. parallel composition: it is not a "behavioural" equivalence for parallel composition. Two sequential designs are behaviorally-equivalent if they produce the same output stream when they receive identical input streams, and this equivalence holds cycle-by-cycle. A related result has already been shown incorrect by Baclet and al. in [11]. The problem we address may be described as follows. Consider two synchronous components C_1 and C_2 that are composed in cascade composition as shown in Figure 1. The output of component C_1 feeds the inputs of C_2. A reaction of C consists of reaction of both C_1 and C_2: C_1 reacts first, produces its output (if any) and C_2 reacts. The goal is to replace the block C_1 and C_2 within the system by the component C such that the resulting system is behaviorally equivalent to the initial one. Baclet and al. show that building the overall system by using the classical composability definition of I/O-models does not guarantee the correctness of the result. This composability does not preserve the behavioural equivalence.

The lack of behaviour preservation makes the reusability inadequate for system development. In particular, to rigorously develop safety-critical systems the preservation of behaviour is a major concern.

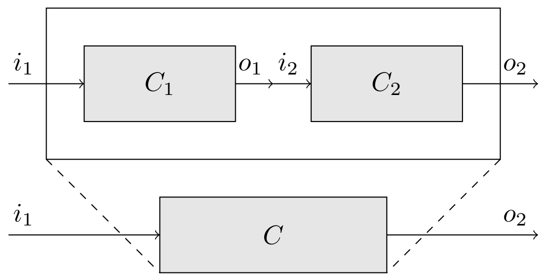


Fig. 1: Cascade composition of two components

The main contribution of this paper is the definition of behaviorally-preserving composition. Specifically, we propose a solution for the composition problem that is able to cope with composability of SR-models. We define novel rules for modelling and composing synchronous components; these rules allow the formal verification of composability. The paper flow is as follows: we introduce in Section II the notion of synchronous component, the model of synchronous component, and the definition of synchronous composition. Section III contains the main contribution of the paper; it presents the problem of composability by using existing composition rules. It presents also a solution: new compositional rules that can give a guarantee of correctness. In Section IV, we present existing approaches that dealt with synchronous composition. Section V concludes the paper and discusses the future perspectives of our work.

II. Synchronous-Reactive Systems

In circuit design, the term synchronous refers to a style of design where a clock that is distributed throughout a circuit drives the execution. The design of Synchronous-Reactive system (SR system for short) is based on synchronous style, in which all components react simultaneously and instantaneously at each tick of a global clock. A SR-system interacts with another by exchanging data. During an execution of a SR system, the system may consume data from input, produce data on output. The systems we considered produce and consume a fixed number of data items per firing, and each output depends on one input data items. SR-systems are described by two parameters: latency and memory. The *latency* is the delay required to process a single data from input to output. Latency is expressed in time units, a suitable unit can be chosen at implementation level depending on the target. The *memory* is the maximum amount of information that can be stored, i.e, storage capacity of the system.

SR-systems we consider have two important properties: determinacy and receptiveness. SR systems are deterministic and receptive. In fact, the receptiveness ensures the interaction between the system and its environment. Input from its environment is never blocked by a system. Symmetrically, output from a system is never blocked by its environment [9].

A. Modelling

Input/Output automaton (I/O-automaton for short) [10] is a well-known mathematical model for SR-systems. I/O-automaton is a state machine at each reaction, it receives as input a data from the environment and produces an output that specifies the new value of the consumed data (which may be the same as the input value).

Definition 1 (I/O-automata): An I/O-automaton is a tuple $\mathcal{A} \triangleq \langle S, s_0, \Sigma, \rightarrow \rangle$ where:

- S is a set of states.
- $s_0 \in S$ is the initial state.
- Σ is the set.
- $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation.

Transitions: Each transition $(s, l, s') \in \rightarrow$ is denoted as usual $s \xrightarrow{l} s'$. When no transition is possible for a state is denoted $s \not\rightarrow$.

The execution of all transitions is simultaneous, instantaneous, and it occurs at ticks of the global clock. Each execution performs an action, $a \in \Sigma$ such that $\Sigma = \{i/o, i/\bar{o}, \bar{i}/o, \bar{i}/\bar{o}\}$. This set consists of input/output events tested for their presence or absence at each tick the clock. Intuitively, i/o means an input and an output occur simultaneously; $i/\bar{o}$ means an input occurs, but no output occurs; $\bar{i}/o$ means no input occurs but output occurs; and $\bar{i}/\bar{o}$ no input and no output occur.

Notations: We will use the notation $\pi = s_0 s_1 s_2 \ldots$ to denote a *path* in $\mathcal{A}$ which is a finite or infinite sequence of states starting from the initial state. We denote by $\hat{\pi}$ a simple path to refer to a path which contains no repeated states. $|\pi| \in \mathbb{N}$ denotes the length of π, and for a position $i < |\pi|$ we define a path fragment from position i as $\pi[s_i] = s_0 \ldots s_i$. We denote $\Pi(s)$ the set of all path fragments $\pi[s]$. To count number of transitions matching a given label α we define the following function:

$$\tau(s_i, \alpha, s_{i+1}) = \begin{cases} 1 & \texttt{if } (s_i, \alpha, s_{i+1}) \in \rightarrow \\ 0 & \texttt{otherwise} \end{cases}$$

An SR-model is an I/O-automaton that satisfies determinacy and receptiveness properties. It satisfies also some behavioural rules which describe when and how the model reacts, according to the value of inputs and its current state. These rules that can be used, as safety properties, to help prove the correctness of an SR-model are the following:

- The model accepts data from the environment as long as it has storage capacity.
- The data wait the duration of the latency before output.
- Firing a transition corresponds to a unit of time elapsed. The latency is calculated by counting the number of transitions. Idleness is allowed only at the end of counting. More precisely, the system stays in the same idle-state, if the total elapsed time allows accomplishment of at least two outputs.
- Once the data is ready to go out, the model has the choice of outputing or waiting.
- Two states which consumed the same amount of data and which can produce same flow results, i.e, the same throughout, are considered equivalent. So to avoid the states explosion problem of generated SR-model, for the set of the equivalent states, we generate only their representative: the state which has the shortest path from the initial state.
- Suppose that each consumed data consists of a time stamp. Thus the entirely of data consumed consists of a sequence of data placed in time, along a real time line. The representative state of the equivalent states will be a bounded sequence (bounded by latency value).

Consider a SR-system and its corresponding SR-model $\mathcal{A}$, the parameters *Latency* denoted by $\mathtt{L}$ and *Memory* denoted by $\mathtt{M}$ are defined formally as follows.

Definition 2 (Latency): Latency $\mathtt{L}$ is the size of the longest path π of $\mathcal{A}$ such that $\forall i < |\pi|, \nexists s' \in S.\, s_i \xrightarrow{\alpha/o} s'$.

Definition 3 (Memory): Memory $\mathtt{M}$ is the size of the longest path π of $\mathcal{A}$ such that $\forall i < |\pi|, s_i \xrightarrow{i/\bar{o}} s_{i+1}$.

Given a state s we denote by $m(s)$ the number of data consumed by the component when it fires. It is calculated by a maximum number of transitions labelled by $i/\bar{o}$ from s_0 to s. More formally:

$$m(s) = max \bigcup_{\pi \in \Pi(s)} \Big\{ \sum_{s_i = s_0}^{pre(s)} \tau(s_i, i/\bar{o}, s_{i+1}) \Big\}$$

where *pre*(s) refers to a previous state of the state s.

The safety property, stating that "it is never possible to overflow a memory" holds in every state. It is expressed: $\forall s \in S.m(s) \leq \mathtt{M}$.

Staying in the same state with the idle action, waiting until new input or output events occur is specified:

$$\left(\begin{array}{l} (s = s_0) \vee \\ (\exists s' \in S.s \xrightarrow{\bar{i}/o} s') \wedge \\ (s' = s_0 \vee \exists s'' \in S.s' \xrightarrow{\bar{i}/o} s'') \end{array} \right) \Rightarrow (s \xrightarrow{\bar{i}/\bar{o}} s)$$

Intuitively, this property says that the system will stay in state s only if there is no data or only one data available, or if there are at least two data which are ready to output.

Example 1: Consider an SR-system with $\mathtt{M}=2$ and $\mathtt{L}=2$, the corresponding automaton is depicted graphically in Figure 2.

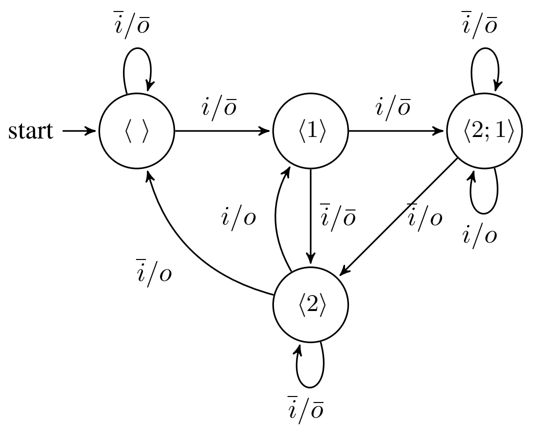


Fig. 2: SR-model of system having $\mathtt{M}=2$ and $\mathtt{L}=2$

For convenience, each state is labelled by a time-stamp (see Figure 2) corresponding to the number of consumed data at that state and for each data the amount of time spent waiting for the output. The initial state $\langle\rangle$ corresponds to the component when it is empty. In this state, two transitions are possible : $(\bar{i}/\bar{o})$ is a passive waiting, it stays in the same state $\langle\rangle$, the second transition $(i/\bar{o})$ leads to the state $\langle 1\rangle$, which can accept an input $i/\bar{o}$ that leads to the $\langle 2;1\rangle$, or have an active waiting to reach the latency ($\mathtt{L}=2$), $\bar{i}/\bar{o}$ to $\langle 2\rangle$. Only in two states $\langle 2\rangle$ and $\langle 2;1\rangle$, is an output possible (i/o), $(\bar{i}/o)$. Both can continue waiting $\bar{i}/\bar{o}$ and stay in the same states. $\langle 2\rangle$ can accept a new input without output $i/\bar{o}$ that leads to $\langle 2;1\rangle$, where the component is full and cannot accept a new input.

B. Composition

Parallel composition of a set of I/O-automata $\mathcal{A}_1 \dots \mathcal{A}_n$, denoted $\mathcal{A}_1 \| \dots \| \mathcal{A}_n$ is defined formally as:

Definition 4 (Composition): Composition of I/O-automata $\mathcal{A}_1 = \langle S_1, s_{0_1}, \Sigma, \rightarrow_1\rangle$ and $\mathcal{A}_2 = \langle S_2, s_{0_2}, \Sigma, \rightarrow_2\rangle$ is the following I/O-automata: $\mathcal{A}_1 \| \mathcal{A}_2 = \langle S_1 \times S_2, (s_{0_1}, s_{0_2}), \Sigma, \rightarrow\rangle$ with $S_1 \times S_2$ the set of states, Σ the set of labels, (s_{0_1}, s_{0_2}) the initial state and $\rightarrow$ is given by the following rules:

$$\rightarrow \triangleq \begin{cases} (s_1,s_2) \xrightarrow{\alpha_1/\beta_1} (s'_1,s'_2) & \texttt{if } (s_1 \xrightarrow{\alpha_1/\alpha^*} s'_1 \wedge s_2 \xrightarrow{\beta^*/\beta_1} s'_2) \\ \texttt{undefined} & \texttt{otherwise} \end{cases}$$

where α^* and β^* is denoting the handshaking synchronisation, i.e, $\alpha = o$ and $\beta = i$ or $\alpha = \bar{o}$ and $\beta = \bar{i}$. Informally, the definition says a composition can be performed if the partners play the same roles for both absence and presence of events.

The SR-model resulting from composition of two SR-models should obviously satisfy properties and behaviour of SR-system. Unfortunately, the result of composition (given in Definition 4) applied to two SR-models does not satisfy all the SR-system properties. Figure 3 illustrates the problem with this composition on an example (detailed in Example 2). The resulting model from a composition may not be a well-formed model. Therefore, this composability is shown inefficient.

The main problem is that the parallel composition rules (Definition 4) appear to be based on stream processing. They put in place data transfer within the resulting model in a pipeline style which is not the baseline behavior of the corresponding component. This composition arises issues that are intrinsic to pipelining: retiming, adding delays and increasing of latency. Actually, the problem occurs with certain configurations or states of its operands, their composition does not allow obtainment of the canonical behavior of the pipeline; and they cause pipeline hazards. Specifically, they produce a model that automatically inserts pipeline bubbles and untidy timeline.

In order to avoid the generation of unwanted configurations and to explain formally the problem, we will introduce flow phenomenons occurring pipelines.

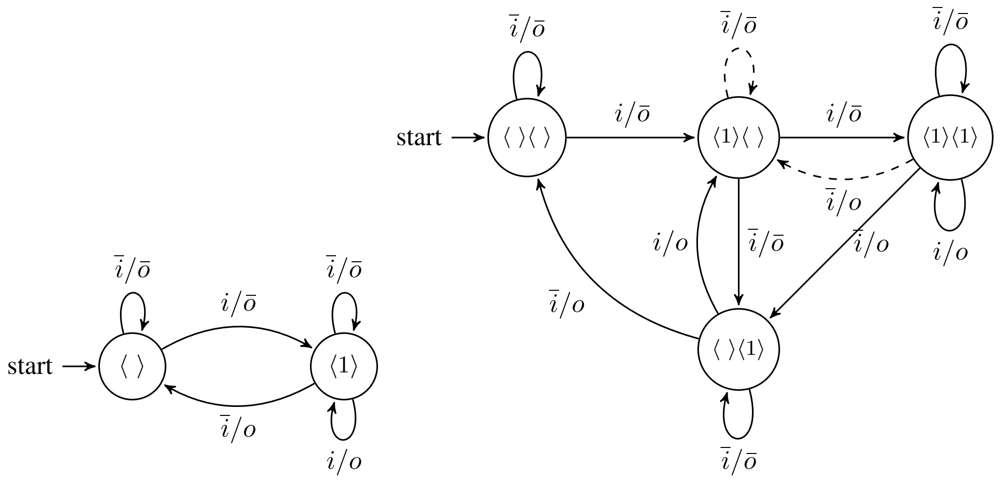
A system is said to be *bubbly*, if it inserts stalls and bubbles in the data flow. Intuitively, a system which accepts and stores data as long as there is available memory and propagates down all data at the same time, then no bubbles can be inserted in it. Thus, we can state that all states of an SR-model satisfies non-bubbly property, this is denoted by $\forall s \in S.\neg \mathit{bubble}(s)$.

A system is said *untidy*, if it puts data flows into a messy state or the wrong order. Intuitively, a system timestamps each arrival of data with the clock. If data are consumed one by one, then data can be tagged with a unique timestamp. Then, the input stream within the system is timestamped and ordered. Thus we can state that all states of an SR-Model satisfies non-untidy property, this is denoted by $\forall s \in S.\neg \mathit{untide}(s)$.

Example 2: Consider a SR-system having the parameters $\mathtt{M}=1$ and $\mathtt{L}=1$, the corresponding SR-Model is given in Figure 3 (left). Parallel composition of this model with itself by applying rules of Definition 4 produces an SR-model given in Figure 3 (right). Serial composition of two SR-systems with $\mathtt{M}=1$ and $\mathtt{L}=1$ should build a SR-system such that $\mathtt{M}=2$ and $\mathtt{L}=2$ ($\mathtt{M}=\mathtt{M}_1+\mathtt{M}_2$ and $\mathtt{L}=\mathtt{L}_1+\mathtt{L}_2$). However, the resulting composed model (given in Figure 3 (right)) is slightly different from what we expected (see Figure 2). Moreover, it is behaviorally incorrect since this automaton is not deterministic. The composition produces two unexpected transitions (see dashed arrows outgoing from the states denoted $\langle 1\rangle\langle\rangle$ and $\langle 1\rangle\langle 1\rangle$). These undesired transitions result from the ability to apply the rules $(\langle 1\rangle \xrightarrow{\bar{i}/\bar{o}} \langle 1\rangle \wedge \langle\rangle \xrightarrow{\bar{i}/\bar{o}} \langle\rangle)$ and $(\langle 1\rangle \xrightarrow{\bar{i}/o} \langle\rangle \wedge \langle\rangle \xrightarrow{i/\bar{o}} \langle 1\rangle)$.

III. Towards safe composition

We have previously shown that the application of composition rules given in Definition 4 generates unwanted states or transitions. These undesired configurations produce an unexpected or incorrect behaviour. This section will discuss and formalize the system configurations that prevent the resulting model from reaching correctness. For this, we need



start
$\langle\,\rangle$
$\langle 1\rangle$
$\bar{i}/\bar{o}$
$i/\bar{o}$
$\bar{i}/o$
i/o
start
$\langle\,\rangle\langle\,\rangle$
$\langle 1\rangle\langle\,\rangle$
$\langle 1\rangle\langle 1\rangle$
$\langle\,\rangle\langle 1\rangle$

Fig. 3: SR-Model with `M`$=1$ and `L`$=1$ (left) and SR-model resulting from composition (right)

to introduce some vocabulary and notations. SR-systems may have several different states, each state with its own properties. An SR-system in state s is said to be in:

- *full* state if the system reached full memory capacity.

$$\textit{full}(s) \Leftrightarrow \nexists s' \in S.s \xrightarrow{i/\bar{o}} s'$$

- *ready* state if the system is willing to deliver outputs.

$$\textit{ready}(s) \Leftrightarrow \exists s' \in S.s \xrightarrow{\alpha/o} s'$$

In order to define unexpected states we introduce also useful predicates for reasoning about flow-related properties of states. An SR-system in state s is said to be in:

- *steady* state if $m(s) = \texttt{L}$. We specify the steadiness on SR-model by:

$$\textit{steady}(s) \Leftrightarrow s \xrightarrow{i/o} s \wedge \exists s' \in S.s \xrightarrow{\bar{i}/o} s' \wedge s' \not\xrightarrow{i/o} s'$$

  it means there is exactly one data ready to go out. In this state, simultaneous consumption and production of data keep the system in the same state.
- *frozen* state if $m(s) > \texttt{L}$. We specify the frozeness on SR-model by:

$$\textit{frozen}(s) \Leftrightarrow s \xrightarrow{i/o} s \wedge \exists s' \in S.s \xrightarrow{\bar{i}/o} s' \wedge s' \xrightarrow{i/o} s'$$

  it means there are more than one data is waiting to go out. The effects of new inputs are no longer important, implying the timeline of data in the system is full.
- *delayed* state if $m(s) < \texttt{L}$. We specify the lateness on SR-model by:

$$\textit{delayed}(s) \Leftrightarrow s \not\xrightarrow{i/o} s$$

  it means data consuming is slow, and the system doesn't reach steady state.

What is particularly interesting about steady-state and frozen-state is that the timeline of consumed data stream at this stage is continuous, it means the timeline is an arithmetic sequence with common difference of 1. We denote a state s satisfying this arithmetic sequence property by *seq*(s). Note that *seq*(s) is slightly different from *untide*(s). For both data stream is a sequence of data that are in order. But for the first, the sequence has a rule to find the value of each data.

It is obvious that every state of the system satisfies exactly one of these flow-related properties at the same time. In addition, we have $\neg\textit{frozen}(s) \Leftrightarrow \textit{steady}(s) \vee \textit{delayed}(s)$, $\neg\textit{delayed}(s) \Leftrightarrow \textit{steady}(s) \vee \textit{frozen}(s)$ and $\neg\textit{steady}(s) \Leftrightarrow \textit{frozen}(s) \vee \textit{delayed}(s)$.

a) Misbehavior analysis: we analysis various configurations of interest to survey the usual composition of SR-models in order to identify causes of misbehavior.

As discussed in the example given above (Example 2), the composition rules, as defined (Definition 4), lead to undesirable non-determinism. This undesirable result for synchrony modelling is caused by configurations for which both composition rules are applicable and are applied. More precisely, the problem occurs when both the following rules are applied:

$$\begin{cases} \left(s_1 \xrightarrow{\alpha/o} s_1' \wedge s_2 \xrightarrow{i/\beta} s_2'\right) \quad \texttt{then } (s_1,s_2) \xrightarrow{\alpha/\beta} (s_1',s_2') \\ \textbf{and} \\ \left(s_1 \xrightarrow{\alpha/\bar{o}} s_1'' \wedge s_2 \xrightarrow{\bar{i}/\beta} s_2''\right) \quad \texttt{then } (s_1,s_2) \xrightarrow{\alpha/\beta} (s_1'',s_2'') \end{cases}$$

The problem happens when both rules (enabled and disabled handshaking) are possible. It means when s_1 and s_2 satisfy $(\textit{ready}(s_1) \wedge \neg\textit{full}(s_2))$. According to the behavioural convention: once the data is ready to go out, the model has the choice of outputing or waiting, component one may produce a data or not. So there are two possible rules to apply.

We identified another types of configuration causing misbehavior. In this type, the configurations generate states from which expected next states are never reached. This is due to the inability to apply any more composition rules on the obtained state. There are several of these kinds of states. Consider a composite state (s_1, s_2) that satisfies :

- $\neg$*full*(s_1, s_2). Suppose that this state is generated from s_1 and s_2 that satisfy (*full*$(s_1) \wedge \neg$*full*(s_2)). We know from the SR-model, there should exist (s'_1, s'_2) such that $(s_1, s_2) \xrightarrow{i/\beta} (s'_1, s'_2)$. But, if s_1 satisfies $\neg$*ready*(s_1), then (s'_1, s'_2) will never be generated. Actually, it can be generated by applying either:
  $(s_1 \xrightarrow{i/o} s'_1) \wedge (s_2 \xrightarrow{i/\beta} s'_2)$ or $(s_1 \xrightarrow{i/\bar{o}} s'_1) \wedge (s_2 \xrightarrow{\bar{i}/\beta} s'_2)$. Nevertheless neither the first nor the second rule can be applied. Actually, as s_1 satisfies $\neg$*ready*(s_1), it means $s_1 \not\xrightarrow{i/o} s'_1$, hence the first rule cannot be matched and as s_1 satisfies also *full*(s_1), it means $s_1 \not\xrightarrow{i/\bar{o}} s'_1$, hence the second rule cannot be matched.
  Actually, (s_1, s_2) is bubbly state, it inserts bubbles in the resulting system. Even if there is still memory available, it cannot accept any data. Bubbly states ere specified as:
  $$\forall s_1 \in S_1, \forall s_2 \in S_2.(\textit{full}(s_1) \wedge \neg \textit{ready}(s_1)) \wedge \neg \textit{full}(s_2) \Rightarrow \textit{bubble}(s_1, s_2)$$
- $\neg$*ready*(s_1, s_2). Suppose that this state is generated from s_1 and s_2 that satisfy (*ready*$(s_1) \wedge \neg$*ready*(s_2)). We know from the SR-model, there should exist (s'_1, s'_2) such that $(s_1, s_2) \xrightarrow{\alpha/\bar{o}} (s'_1, s'_2)$. But if s_2 satisfies *full*(s_2), then $s_2 \not\xrightarrow{i/\bar{o}} s'_2$. Hence $(s_1 \xrightarrow{\alpha/o} s'_1) \wedge (s_2 \xrightarrow{i/\bar{o}} s'_2)$ cannot be applied. Actually, component one is ready to output data, but waits until component two is ready to pick up. This will cause a propagation delay. The time that will take the data to travel from component one (input) to component two (output) will be greater than latency.
  (s_1, s_2) is an untidy state, it is specified as:
  $$\forall s_1 \in S_1, \forall s_2 \in S_2.\textit{ready}(s_1) \wedge (\textit{full}(s_2) \wedge \neg \textit{ready}(s_2)) \Rightarrow \textit{untide}(s_1, s_2)$$
  More generally, an *untide* composite state can be generated since component one is willing to output data, but it does not. Specifically, when we have the following configuration: $(s_1 \xrightarrow{\alpha/o} s'_1) \wedge (s_2 \not\xrightarrow{i/\beta} s'_2)$ or we apply $(s_1 \xrightarrow{\alpha/\bar{o}} s'_1) \wedge (s_2 \xrightarrow{\bar{i}/\beta} s'_2)$.

The last undesirable configuration we consider is the *frozen* composite state, *frozen*(s_1, s_2). This state can be generated from s_1 satisfying any flow-related properties and from s_2 satisfying necessarily *frozen*(s_2). But, if s_1 satisfies *delayed*(s_1) then the result of the composition will be false, (s_1, s_2) doesn't satisfy arithmetic sequence property. Therefore, s_1 and s_2 such that *delayed*(s_1) and *frozen*(s_2) should not be composed. This is specified as:
$$\forall s_1 \in S_1, \forall s_2 \in S_2.\textit{delayed}(s_1) \wedge \textit{frozen}(s_2) \Rightarrow \neg \textit{seq}(s_1, s_2)$$

The idea of the proposed solution is on one part, to prevent the composition of any states s_1 and s_2 that can generate an undesirable composite state (s_1, s_2), namely (s_1, s_2) satisfying *bubble*(s_1, s_2) and *untide*(s_1, s_2); on the other part, to find a way to avoid the problem of missing expected states.

b) Proposed composition: To solve in practice the problem of behavioral correctness of SR-models composition, basic needs will have to be dealt with, namely to give the possibility of consuming data as long as there is available memory. Clearly, to consume when component two is not full even if component one is full. And to carry on with calculating elapsed time for data which are ready to go out in component one, even if component two is not ready to receive them. Our proposal is based on the idea of extension of SR-models of analyzed components. Actually, in order to build global models for the serial-composition of components having respectively the parameters $\mathtt{M}_1$ and $\mathtt{L}_1$, and $\mathtt{M}_2$ and $\mathtt{L}_2$. We construct the parallel composition of extended SR-models instead of their corresponding SR-models. Consider a component having the parameters $\mathtt{M}$ and $\mathtt{L}$, and $\mathcal{A}$ the corresponding SR-model. We denote by $\mathcal{A}^{+\mathtt{m}}$ its extended SR-model with an increase of memory storage capacity by an amount of memory equals to $\mathtt{m}$. Therefore, the storage capacity of the component goes from $\mathtt{M}$ to $\mathtt{M} + \mathtt{m}$. The serial-composition of the two components is defined by $\mathcal{A}_1^{+\mathtt{m}_1} \| \mathcal{A}_2^{+\mathtt{m}_2}$ where:
$$\mathtt{m}_1 = \begin{cases} \textit{min}(\mathtt{L}_1 - \mathtt{M}_1, \mathtt{M}_2) & \mathtt{if}\ \mathtt{M}_1 < \mathtt{L}_1 \\ 0 & \mathtt{otherwise} \end{cases}$$
$$\mathtt{m}_2 = \begin{cases} \mathtt{L}_2 - \mathtt{M}_2 & \mathtt{if}\ \mathtt{M}_2 < \mathtt{L}_2 \\ 0 & \mathtt{otherwise} \end{cases}$$

Referring to Figure 1, the composition that we propose extends the storage capacity of both components C_1 and C_2, so as to be able to solve the problem of data storage and the calculation of latency. The memory of C_1 is extended by amount $\mathtt{m}_1$ so as to receive the data that it was not possible to put in C_2 due to pipeline transport of data. The memory of C_2 is extended by amount $\mathtt{m}_2$ so as to receive the data that C_1 has to bring out but C_2 cannot receive due to the lack of memory space. Notice that if $\mathtt{M}_1 \geq \mathtt{L}_1$ (resp. $\mathtt{M}_2 \geq \mathtt{L}_2$) there is no need to extend C_1 (resp. C_2) because stream is flowing freely and the memory is never full.

Before defining the composition of the extended SR-models, we introduce some notation. We will use a priority operator, denoted $\curlywedge$, for specifying the order in which the rules are applied when a state has several applicable rules. We typically write $\overset{R_1}{\underset{R_2}{\curlywedge}}$ to mean if both rules R_1 and R_2 are applicable we apply R_1, otherwise we apply R_2. We also define a predicate *exceed* that indicates if a component exceeds a given amount of data.
$$\textit{exceed}(s, M) \Leftrightarrow m(s) > M$$

Logically, no state of a well formed SR-model satisfies *exceed*$(s, \mathtt{M})$ with $\mathtt{M}$ the memory size. We will use another predicate that we call *idle* that indicates if a component is willing to bring out data but without emptying.
$$\textit{idle}(s) \Leftrightarrow s \xrightarrow{\bar{i}/\bar{o}} s \wedge \exists s'' \in S.s \xrightarrow{\bar{i}/o} s'' \wedge s'' \neq s_0$$

According to semantics of a SR-model, the idle action can be supported by a state that has either zero or a single data, or at least two data ready to go out. Accordingly, a state s satisfying $idle(s)$ means from s there is a next state that is also willing to produce data.

We now define the composition of extended SR-models.

Definition 5 (Extended Composition): Consider two components C_1 and C_2 with parameters $\mathtt{M}_1$, $\mathtt{L}_1$ and $\mathtt{M}_2$, $\mathtt{L}_2$, respectively. The composition of the corresponding SR-models $\mathcal{A}_1 = \langle S_1, s_{0_1}, \Sigma, \rightarrow_1\rangle$ and $\mathcal{A}_2 = \langle S_2, s_{0_2}, \Sigma, \rightarrow_2\rangle$, written $\mathcal{A}_1^{+\mathtt{m}_1} \| \mathcal{A}_2^{+\mathtt{m}_2}$ is an SR-model $\langle S_1 \times S_2, (s_{0_1}, s_{0_2}), \Sigma, \rightarrow\rangle$ such that $\rightarrow$ is defined by the following rules:

1) $s_1 \xrightarrow{i/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/o} s_2'$
$$\begin{cases} s_1 \xrightarrow{i/o} s_1', s_2 \xrightarrow{i/o} s_2' \\ \quad\curlywedge \\ s_1 \xrightarrow{i/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/o} s_2' \\ \qquad\qquad\qquad (s_1, s_2) \xrightarrow{i/o} (s_1', s_2') \end{cases}$$

2) $s_1 \xrightarrow{\bar{i}/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/o} s_2'$
$$\begin{cases} s_1 \xrightarrow{\bar{i}/o} s_1', s_2 \xrightarrow{i/o} s_2' \quad \texttt{if } \mathit{seq}(s_1', s_2') \\ \quad\curlywedge \\ s_1 \xrightarrow{\bar{i}/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/o} s_2' \\ \qquad\qquad\qquad (s_1, s_2) \xrightarrow{\bar{i}/o} (s_1', s_2') \end{cases}$$

3) $s_1 \xrightarrow{i/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/\bar{o}} s_2'$
$\texttt{if } \neg\,\mathit{exceed}((s_1, s_2), M_1 + M_2) \texttt{ then}$
$$\begin{cases} s_1 \xrightarrow{i/o} s_1', s_2 \xrightarrow{i/\bar{o}} s_2' \quad \texttt{if } \mathit{seq}(s_1', s_2') \\ \quad\curlywedge \\ s_1 \xrightarrow{i/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/\bar{o}} s_2' \\ \qquad\qquad\qquad (s_1, s_2) \xrightarrow{i/\bar{o}} (s_1', s_2') \end{cases}$$

4) $s_1 \xrightarrow{\bar{i}/\alpha^*} s_1', s_2 \xrightarrow{\beta^*/\bar{o}} s_2'$
$\texttt{if } \mathit{idle}(s_2) \texttt{ then} \qquad (s_1, s_2) \xrightarrow{\bar{i}/\bar{o}} (s_1, s_2)$
$\texttt{else}$
$$\begin{cases} s_1 \xrightarrow{\bar{i}/o} s_1', s_2 \xrightarrow{i/\bar{o}} s_2' \quad \mathit{delayed}(s_2) \\ \quad\curlywedge \\ s_1 \xrightarrow{\bar{i}/\bar{o}} s_1', s_2 \xrightarrow{\bar{i}/\bar{o}} s_2' \\ \qquad\qquad\qquad (s_1, s_2) \xrightarrow{\bar{i}/\bar{o}} (s_1', s_2') \end{cases}$$

Intuitively, defining the composition on extended SR-models allows the generation of the missing states. And defining conditions under which the rules apply prevents the generation of the undesirable states. Note the use of the operator priority in each case which specifies the method for applying rules: when both rules are applicable the transfer takes precedence. Otherwise, rules behave roughly as basic rules except that a transfer between two SR-models is allowed only if the resulting composite state guarantees the sequentiality in data flow ($\mathit{seq}(s_1', s_2')$). However, the composition is not allowed for states leading to the memory overflow of the global component. Indeed, due to the additional memory of the extended SR-models, the amount of data in a composite state can exceed the size $\mathtt{M}_1 + \mathtt{M}_2$, so the composition avoids generation of such states ($\neg\,\mathit{exceed}(s_1, s_2)$). The particular case is the generation of the idle action. If the second component is in a state idle ($\mathit{idle}(s_2)$), the global system will also be in idle state. Otherwise, an idle action can be a result of applying composition rules.

We developed in Ocaml programming language, a tool which implements the algorithm of construction of an SR-model from a component defined by its parameters. The composition rules were implemented in the tool. We validated the correctness of the result by using the CADP model checker [8]. We checked the equivalence of the composition result of different test-cases with the expected SR-models, the comparison succeed for all test-cases.

IV. Related work

Component-based development has in recent years become an established approach. It has proven successful in many application domains such as in distributed and embedded systems. There are several component models that are supplied for building complex hardware or software systems: Fractal [5], Ptolemy [6], CCM [14], AADL, BIP [2] and GCM [1]. But there are only a few that have a theoretical framework that allows reasoning about compositional modelling systems, specially about modelling synchronous systems and verification of their behavioural properties. Among the researches dedicated to the component-oriented verification of embedded systems that we are aware of [15], [12], [13], [3], the closest is BIP. The BIP tool provides a formal framework for modeling system behavior and architectures. A component's behaviour is described as a Petri net extended with data and functions, whereas coordination is described as interactions between components and scheduling policies between interactions. Even if the BIP framework allows powerful compositional reasoning on the system, it is shown [4] that the proposed semantics is not sound and the synchrony is weakened to get more synchronous computation models.

V. Conclusion

Through this paper, we presented models for compositional reasoning about complex systems. We have introduced new compositional rules for serial composition of synchronous components, with the hope of making formal verification apply to a wider range of complex designs. We provide rules for compositional design and verification based on I/O automata. We argue that the compositionality of I/O automata has the ability, from an engineering point of view, to allow reasoning about component-based systems of many different kinds, including real-world components, computer programs, communication channels, sensors, etc. Our solution handles stream processing between components, particularly synchronous components and can be applied generally to all SR-models. The reasoning rules have already been implemented and their validity has been verified by using CADP tool. We aim to further improve the verification by proving the equivalence between the result of composition and expected

SR-models. Finally, we would like to take into account further data-relationship by considering one-to-many and many-to-many relationship between input data and output data of components.

References

[1] R. Ameur-Boulifa, L. Henrio, O. Kulankhina, E. Madelaine, and A. Savu. Behavioural semantics for asynchronous components. *Journal of Logical and Algebraic Methods in Programming*, 89:1 – 40, 2017.

[2] Ananda Basu, Bensalem Bensalem, Marius Bozga, Jacques Combaz, Mohamad Jaber, Thanh-Hung Nguyen, and Joseph Sifakis. Rigorous Component-Based System Design Using the BIP Framework. *IEEE Software*, 28(3):41–48, May 2011.

[3] Julien Boucaron and Jean-Vivien Millo. Compositionality of statically scheduled IP. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 200:71–87, February, 2008.

[4] Marius Dorel Bozga, Vassiliki Sfyrla, and Joseph Sifakis. Modeling Synchronous Systems in BIP. In *Proceedings of the Seventh ACM International Conference on Embedded Software*, EMSOFT '09, pages 77–86, New York, NY, USA, 2009. ACM.

[5] Eric Bruneton, Thierry Coupaye, M. Leclerc, V. Quema, and Jean Bernard Stefani. An Open Component Model and Its Support in Java. In *7th Int. Symp. on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, 2004.

[6] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia R. Sachs, and Yuhong Xiong. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[7] Fabrício Fernandes and Jean-Claude Royer. The STSLib project: Towards a formal component model based on STS. *Electronic Notes in Theoretical Computer Science*, 215:131–149, 2008.

[8] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *European Association for Software Science and Technology Newsletter*, 4:13–24, aug 2002.

[9] Mark B. Josephs. Receptive process theory. *Acta Informatica*, 29(1):17–31, 1992.

[10] N. Lynch and M. Tuttle. An introduction to Input/Output automata. *CWI-Quarterkly*, 2(3):219–246, 1989.

[11] R. Pacalet M. Baclet and A. Petit. Register transfer level simulation. *Research Report LSV-04-10. Laboratoire Spécification et Vérification. ENS de Cachan. France*, may, 2004.

[12] Florence Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *International Conference on Concurrency Theory (CONCUR)*. LNCS 630, Springer Verlag, aug 1992.

[13] Florence Maraninchi and Tayeb Bouhadiba. Programmable models of computation for a component-based approach to heterogeneous embedded systems. *In Proceedings of ACM-GPCE'07.*, 2007.

[14] Object Management Group, Inc. (OMG). *CORBA Component Model Specification*, OMG Headquarters edition, April 2006.

[15] Annie Ressouche Sabine Moisan and Jean-Paul Rigault. Towards formalizing behavioral substitutability in component frameworks. *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, pages 122–131, September 28-30, 2004.