

## Modular Analysis of Dataflow Process Networks

Yan Jin<sup>1</sup>, Robert Esser<sup>1</sup>, Charles Lakos<sup>1</sup>, and Jörn W. Janneck<sup>2</sup>

<sup>1</sup> School of Computer Science, University of Adelaide, SA 5005, Australia  
{yan, esser, charles}@cs.adelaide.edu.au

Fax: +61 8 8303 4366

<sup>2</sup> EECS Department, University of California at Berkeley, CA 94720-1770, USA

jwj@acm.org

Fax: +1 510 642 2739

**Abstract.** Process networks are popular for modelling distributed computing and signal processing applications, and multi-processor architectures. At the architecture description level, they have the flexibility to model actual processes using various formalisms. This is especially important where the systems are composed of parts with different characteristics, *e.g.* control-based or dataflow-oriented. However, this heterogeneity of processes presents a challenge for the analysis of process networks. This research proposes a lightweight method for analysing properties of such networks, such as freedom from unexpected reception and deadlock. The method employs interface automata as a bridge between the architectural model and heterogeneous processes. Thus, the properties are determined by a series of small tasks at both the architecture level and the process level. This separation of concerns simplifies the handling of heterogeneous processes and alleviates the potential state space explosion problem when analysing large systems.

### 1 Introduction

In recent years, component-based development has emerged as a significant factor in the production of large-scale software applications. By building systems from independently developed components, a promising means of achieving software reuse, rapid development and complexity management is provided.

Typically, components are black-box entities that encapsulate services behind their interfaces. The specifications of these interfaces tend to be rather limited, often capturing only the *signatures* of components, *i.e.* the names, data types and direction of ports excluding any information about communication *protocols*. Even with additional informal descriptions, such specifications are not adequate for designing reliable and evolving software systems. Instead, more rigorous specifications are needed, which capture interface behaviours of components, including the services that a component provides, the information about how it can be properly deployed, and the dependencies between its inputs and outputs. Naturally these specifications must not disclose implementation details of their components.

Having suitable specifications for the components is only part of the story — it is also necessary to provide flexible composition schemes. Direct composition is often difficult and sometimes impossible [1]. Instead, it is preferable to provide flexible

connectors so that component-based systems can be constructed using various design strategies [2], where suitable architectural styles, *e.g.* pipe-and-filter and client/server architectures, can be employed. The major challenge is then to ensure that the resulting systems are consistent (namely, all components are properly deployed in the design) and that these systems meet global functional and nonfunctional requirements such as structural invariants, reliability and security.

This paper presents a step towards the component-based development and modular analysis of dataflow process networks. Such networks support flexible interconnection strategies as mentioned above. In our presentation, components (or processes) communicate through their input and output ports and the interconnection of components specifies a causality relation between data flow through input/output ports of the components.

In order to avoid the state space explosion problem, the composition of components is not analysed directly. Instead, interface automata [3] are used to specify not only the interface behaviour of components but also their assumptions about the environment. Abstracting away implementation details of components, interface automata are much simpler and easier to handle. Firstly, we ensure that each component is consistent with its corresponding interface automaton, namely, each component is able to fulfil the output guarantee of the automaton under the environmental assumptions of the automaton.

Secondly, we check the consistency of the network comprising these interface automata. This consistency can in turn justify that the process network is free from unexpected reception and deadlock. The former property indicates that the data flow between its components is directed in a way which is consistent with the assumptions made by the components. The latter refers to the absence of deadlock where the system cannot make any further progress. By adopting such a divide-and-conquer approach, the potential state space explosion problem can be alleviated.

The presented research is motivated by previous work on the Moses tool suite [4]. Moses presents an additional challenge to component-based development in that it supports the modelling and simulation of heterogeneous discrete-event systems, where components are modelled by different formalisms. For example, components can be defined [4, 7, 9] as process networks, Petri Nets, Statecharts, etc. The proposed approach can also largely simplify the analysis of such heterogeneous systems.

This paper is structured as follows. In section 2, our approach is compared with the related work. In section 3, we define discrete-event components (or processes), interface automata and the consistency between them, and present a practical method for checking the consistency. In section 4, we define dataflow process networks and interface automaton networks and present our method of modular analysis of dataflow process networks. Finally, we conclude this paper in section 5.

## 2 Related work

Interface automata were first introduced in [3]. The authors established a simple and well-defined semantics for them and defined their composition by two-party synchronization. Also, alternating simulation was proposed to determine a refinement relationship between interface automata. This relationship takes an optimistic view of the envi-

ronment by assuming that it is always helpful, only supplying inputs expected by an automaton. This optimistic view allows more possible implementations than a pessimistic approach where the environment can behave as it pleases.

We take this optimistic view and adapt alternating simulation to define the consistency of components with interface automata, taking into account data values used in components. In order to prove properties such as deadlock freedom, an additional restriction is imposed which requires a component to produce at least one of the outputs that the corresponding interface automaton can produce. Also, a simpler method of checking the relation is presented, which does not require the construction of the Cartesian product of their state spaces as [3] does – only the reachable states in the product need to be constructed. Additionally, in contrast to the simple composition scheme in [3], we allow interface automata to be composed in many more ways reflecting how process networks can be constructed in practice.

Our adaptation and checking method of alternating simulation is inspired by [13], where a similar relation was proposed to check the conformance (or refinement) relationship between CCS models. However, this relation is more restricted than ours, as it requires that both specification and implementation models have no mixed states (where both input and output transitions can occur), while in our approach this is only required of the specification, *i.e.* interface automata. Furthermore, because of the presence of blocking outputs, models in their approach are different in nature from ours where a component is in full control of its outputs. In addition, in our approach the substitutability of heterogeneous components can be checked with the aid of interface automata.

There are some other approaches which also utilize the environmental assumptions of components for verification. In [5, 6], the assumptions and the actual behaviours of components are derived from component specifications. The deadlock freedom of a system is determined by pairwise matching between the assumptions of a component and the actual behaviour of another component. However, the proposed method is incomplete and limited to one-to-one communication or synchronization.

The approach in [16] requires additional models of the environmental assumptions of components. The models are used to restrict the behaviour of the environment so that system deadlock can be discovered by detecting undesirable usage of components. However, the global state space needs to be built, which would easily lead to the state space explosion problem.

### 3 Consistency of components

In this section, general reactive systems are first introduced. These are specialised as discrete-event components in section 3.2 and as interface automata in section 3.3. In section 3.4, the consistency of discrete-event components with interface automata is defined. A practical method of consistency checking is presented in section 3.5.

#### 3.1 Reactive transition systems

**Definition 1.** A reactive transition system (RTS) is defined as  $L = (s^0, S, \Sigma, \rightarrow)$ , where

- $S$  is a set (possibly infinite) of states and  $s^0 \in S$  is the initial state;
- $\Sigma$  is a set (possibly infinite) of events, consisting of three mutually disjoint sets of input events  $\Sigma^I$ , output events  $\Sigma^O$ , and internal events  $\Sigma^H$ ;
- $\rightarrow \subseteq S \times \Sigma \times S$  is a set of transitions.

This definition draws an explicit distinction between input, output and internal events. This is because a system has control over its internal and output events only, but no control over its input events. Instead, when an input event occurs is decided by the environment. In other words, the system cannot prevent the environment from producing an input event if it wants to do so. In the following, we let  $\Sigma^{obs} = \Sigma^I \cup \Sigma^O$  be a set of observable events and  $\Sigma^{ctl} = \Sigma^O \cup \Sigma^H$  be a set of controllable events of  $L$ .

**Definition 2.** A trace  $\sigma$  of a RTS  $L$  from  $s_1 \in S$  is an event sequence  $e_1 e_2 \dots e_m$  such that  $\forall j: 1 \leq j \leq m, \exists (s_j, e_j, s_{j+1}) \in \rightarrow$ . State  $s_{m+1}$  is called *reachable* in  $L$  (via  $\sigma$ ) if  $\sigma$  is from  $s^0$ . Also, a trace is said to be *internal* if  $\forall j: 1 \leq j \leq m, e_j \in \Sigma^H$  and to be empty if  $m = 0$ . An empty trace is denoted as  $\lambda$ .

The restriction  $\sigma \upharpoonright E$  of  $\sigma$  on an event set  $E$  is an event sequence obtained by removing from  $\sigma$  all events not in  $E$ .

In the sequel, we write  $s \xrightarrow{e} s'$  to denote  $(s, e, s') \in \rightarrow$ . We also write  $s \Longrightarrow s'$  if  $s'$  is reachable via a (possibly empty) internal trace of  $L$  from  $s$ , and  $s \xrightarrow{e} s'$  for  $e \in \Sigma^{obs}$  if  $s \Longrightarrow s'' \wedge s'' \xrightarrow{e} s'$ .

A RTS  $L$  is called *deterministic* if  $\forall e \in \Sigma, s, s', s'' \in S, s \xrightarrow{e} s' \wedge s \xrightarrow{e} s''$  implies  $s' = s''$ . It is said to be *nondeterministic* otherwise. Also, the sets of *enabled input and output events* at a state  $s \in S$  are defined by  $\text{en}^I(s) = \{e \in \Sigma^I \mid \exists s' \in S, (s, e, s') \in \rightarrow\}$  and  $\text{en}^O(s) = \{e \in \Sigma^O \mid \exists s' \in S, s \xrightarrow{e} s'\}$ , respectively. An event  $e \in \Sigma^I$  is said to be *refused* at  $s$  if  $e \notin \text{en}^I(s)$ .  $L$  is said to be *input-universal* if  $\forall s \in S, \text{en}^I(s) = \Sigma^I$ . Additionally, a state  $s$  is called a *terminal state* if  $\nexists (s, e, s') \in \rightarrow$ .

### 3.2 Discrete-event components

A Moses component is a discrete-event component that consumes data streams fed to its input ports, and produces data streams through its output ports. The input and output ports form a component's view of the rest of the system and decouple the outside world from the component. This separation allows the behaviour of the component to be described independently of its ultimate context. Likewise, the outside world learns about a component only from the communication through its ports. In defining components, we assume a universal set  $\mathcal{U}^{port}$  of ports, and a countable universal set  $\mathcal{U}^{val}$  of values of data flowing through ports.

**Definition 3.** Given two disjoint finite sets of its input ports  $\alpha^I \subset \mathcal{U}^{port}$  and its output ports  $\alpha^O \subset \mathcal{U}^{port}$ , a discrete-event component (DEC) is defined as an input-universal RTS  $C = (s^0, S, \Sigma, \rightarrow)$ , where  $\Sigma^I = \alpha^I \times \mathcal{U}^{val}$ ,  $\Sigma^O \subseteq \alpha^O \times \mathcal{U}^{val}$  and  $\Sigma^H$  is a set of fresh labels for transitions with no external effect.

In the definition, an input/output event of  $C$  is regarded as an occurrence of data flowing through an input/output port of  $C$ , while internal events of each DEC are considered to be unique to that DEC. Also, a DEC is an input-universal RTS, i.e. it never

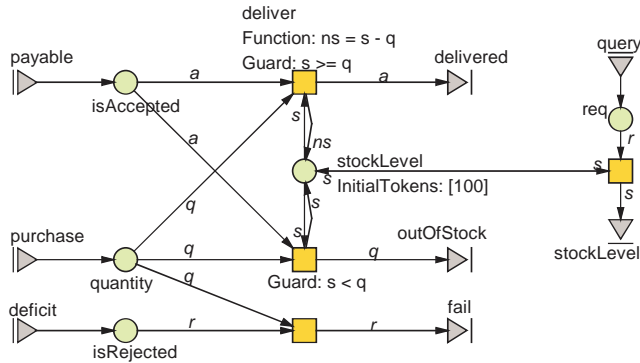


Fig. 1. A store component

refuses an input and hence writing to a component will never block. Typically, each component has one or more input buffers (generally of infinite length), which are either implicit or explicit depending on the modelling language. For instance, a Petri Net component may have multiple places acting as explicit buffers [7], while a UML Statechart component has only one implicit buffer for all input ports [9]. These built-in buffers ensure the input acceptance of components. In the sequel, we assume a universal set  $\mathcal{U}^{dec}$  of DEC's and ports of each DEC to be unique to the DEC.

Figure 1 gives an example of a Petri Net component in Moses, where triangles represent the input/output ports of components and where the body of the component is given in the usual Petri Net notation with circles, boxes and arcs representing places, transitions and the flow relationships, respectively. When data comes into a component via its input port, it is added to the place(s) connected to this port. A transition (e.g. “deliver” in figure 1) becomes enabled once all its input places have enough tokens and its guard evaluates to true. As is the case for other high-level Petri nets (e.g. [8]), this binds the tokens to the variable names (e.g. “a”, “q” and “s”) on its incoming arcs, and finally the transition fires. While firing, the transition binds the variable names (e.g. “ns” and “a”) on the outgoing arcs depending on the values of the variables on the incoming arcs. When a firing transition is connected to an output port (e.g. “delivered”), data is sent out via the port to all connected components<sup>3</sup>. For a Petri Net component, we require input ports to be connected to places and output ports to transitions to ensure that any component output can be meaningfully connected to any component input. Assuming the interleaving semantics of Petri Net components [7], the interpretation of such components in terms of discrete-event components is straightforward and we omit this for the sake of brevity. See [14] for the basic concepts of Petri Nets, and [4, 7] for more descriptions of the Moses approach to compositional Petri Nets.

This example models an online store that waits for a purchase request from a customer and payment acceptance from the customer’s bank before delivering the goods.

<sup>3</sup> Ports can be considered to segment arcs into three – that part of the arc prior to the output port, that part between output and input port(s), and that part following the input ports.

If the bank refuses to pay or the goods are out of stock, the request fails. The store also reports the stock level when being queried. Initially, the store holds 100 pieces of goods. A successful order will result in the stock level being decreased by the ordered quantity.

### 3.3 Interface automata

Usually, a component is designed under some environmental assumptions about how the component can be properly deployed, *e.g.* interaction protocols. The assumptions are useful for analysing the behaviour of the component, especially when the component is independently developed and analysed. However, an input-universal component cannot constrain the environment as to when and what kind of input to provide. Therefore, these assumptions cannot be captured by component models. In this approach we employ interface automata [3] to solve this problem.

**Definition 4.** An interface automaton<sup>4</sup> (IA) is defined as a deterministic RTS  $A = (s^0, S, \Sigma, \rightarrow)$ , where  $S$  and  $\Sigma$  are finite and  $\Sigma^H = \emptyset$ .

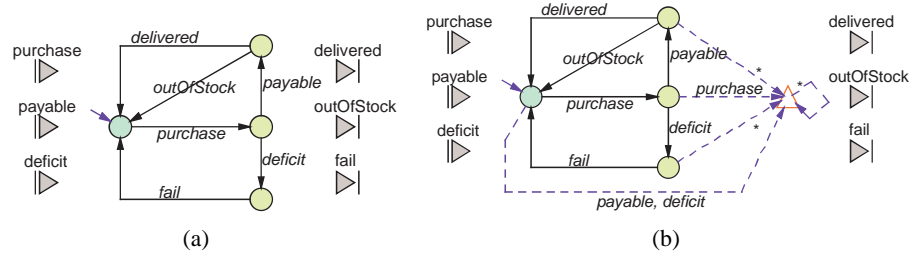
Like [13, 17], we exclude IAs with mixed states, *i.e.* states where both input and output transitions can occur. Also, we assume a universal set  $\mathcal{U}^{ia}$  of IAs.

The information captured by an interface automaton is twofold. On the one hand, the behaviour of the automaton is observed through a sequence of its output events. On the other hand, the assumption is implicitly captured that the environment should never provide an input if the automaton is in a state where the input is refused. Also, when the automaton wishes to produce an output, the environment should be ready to accept it.

As an example, suppose that we have the automaton in figure 2(a) as the interface specification of the store component of figure 1. This captures the assumption that the environment cannot provide a second purchase request before the first one has been processed. Also, after the store receives a purchase request, the environment can either provide a “payable” message indicating that the customer can pay for the purchase or a “deficit” message indicating otherwise. On the other hand, it guarantees that the store produces either a “delivered” or “outOfStock” message but definitely not a “fail” message immediately after receiving a “payable” message.

Definitions 3 and 4 indicate a similarity in behaviour between interface automata and discrete-event components. Here, we consider an input event of an IA corresponds to an occurrence of data flow with an arbitrary value through an input port of a component. Similarly, an occurrence of data flow with an associated value at an output port of the component corresponds to an output event of the IA. In other words, when relating the behaviour of IAs and DEC, the events of the IAs correspond to an abstraction of the events of the DEC, an abstraction which ignores the data values. While abstracting away the implementation details of components, the high-level interface specifications can help simplify the analysis of process networks. They are also very useful in architecture analysis since component models are often not available when designing system architectures.

<sup>4</sup> Originally defined in [3], IAs can have internal events and be nondeterministic. We believe that definition 4 is sufficiently expressive for our purpose.



**Fig. 2.** A store automaton (a) and its input-universal RTS (b)

### 3.4 Consistency of discrete-event components

The association of interface automata with discrete-event components leads to the most important issue in this approach, that is, the consistency of DEC with IAs. The consistency refers to the fact that an IA can safely be substituted by a DEC without compromising the properties which previously hold.

The consistency cannot be defined by traditional refinement relations, *e.g.* trace containment and simulation. This is because these only allow the implementation to have less input and output behaviour than the specification, whereas input-universal components are able to handle more inputs than IAs. Hence, we adopt alternating simulation [3] to define consistency.

Alternating simulation is concerned with the relation of an IA with a (helpful) environment. It can be considered as a two-person game, where the automaton will try to perform some action which will cause the environment to block and the environment will try to respond so that the automaton does not succeed in its attempt. Thus, the environment can limit the behaviour of the automaton by not offering certain inputs and the automaton can make things easier for the environment by not generating certain outputs. If an environment is helpful enough for an automaton, then it should also be helpful for a refinement of the automaton. The refinement can offer less outputs (since this will not make as many demands on the environment) and accept more inputs (since the environment will not offer them).

Originally, alternating simulation was used to define the refinement between two IAs, where no data values were involved [3]. We extend this refinement relation to accommodate the implementation (or DEC) with data values. Also, in order to prove properties like deadlock freedom, an additional restriction is imposed which requires the component to generate at least one of the outputs that the automaton can possibly produce.

**Definition 5.** Consider an IA  $A$  and a DEC  $C$  such that  $\Sigma_A^I \subseteq \alpha_C^I$  and  $\Sigma_A^O \supseteq \alpha_C^O$ .  $C$  simulates  $A$ , written  $C \preceq A$ , if there exists a relation  $\preceq \subseteq S_C \times S_A$  such that  $s_C^0 \preceq s_A^0$  and for  $q \preceq s$ , the following conditions hold:

1.  $\text{en}_A^O(s) \neq \emptyset$  implies  $\text{en}_C^O(q) \neq \emptyset$ ;
2.  $\forall \langle f, v \rangle \in \text{en}_C^O(q) \cup (\text{en}_A^I(s) \times U^{val})$ ,  $q \xrightarrow{\langle f, v \rangle}_C q'$  implies  $\exists s' \in S_A$ ,  $s \xrightarrow{f}_A s' \wedge q' \preceq s'$ .

Basically,  $C$  simulates  $A$  if  $C$  is able to fulfil the output guarantee of  $A$  when the environment provides  $C$  only enabled inputs of  $A$ . In other words, the environment provides an input to  $C$  at a state  $q \in S_C$  only when  $A$  at a state  $s \in S_A$  (s.t.  $q \preceq s$ ) is able to accept the input. Also, fulfilling the output guarantee of  $A$  indicates two facts. First, every possible output, which  $C$  can produce at  $q$  or at a state reachable via an internal trace from  $q$ , must also be allowed by  $A$  at  $s$ . Second,  $C$  from  $q$  should be able to produce at least one of the outputs that  $A$  can produce at  $s$ . The definition implies an input and output duality that  $C$  at state  $q$  allows more input events but produces less output events than  $A$  at state  $s$ . It is worth noting that  $\text{en}_C^I(q) \supseteq \text{en}_A^I(s) \times \mathcal{U}^{val}$  always holds for all  $q \in S_C, s \in S_A$ , because  $C$  is input-universal. Note also that condition 2 implies  $\text{en}_C^O(q) \subseteq \text{en}_A^O(s) \times \mathcal{U}^{val}$ .

Definition 5 allows DEC's with equal or less output ports to be the implementation of an IA. However, DEC's often have not only more input ports but also more output ports in practice, especially when third-party components are deployed which may provide more services than needed in an application domain. To solve this, we define instantiated components for these DEC's and relax the conditions of definition 5 in defining the consistency of DEC's with IAs. Note in the following definition that  $\hat{C}(\mathcal{O}) = C$  if  $\alpha_C^O \subseteq \mathcal{O}$ .

**Definition 6.** An instantiated component of a DEC  $C$  with respect to a set  $\mathcal{O} \subseteq \alpha_C^O$  is defined by  $\hat{C}(\mathcal{O}) = (s^0, S, \Sigma_C, \rightarrow_C)$  where  $\Sigma_C^I = \Sigma_C^I, \Sigma_C^O = \{(f, v) \in \Sigma_C^O \mid f \in \mathcal{O}\}$  and  $\Sigma_C^H = \Sigma_C^H \cup \Sigma_C^O \setminus \Sigma_C^O$ .

**Definition 7.** Consider an IA  $A$  and a DEC  $C$  such that  $\Sigma_A^I \subseteq \alpha_C^I$ .  $C$  is consistent with  $A$  if  $\hat{C}(\Sigma_A^O) \preceq A$ .

### 3.5 Practical consistency checking of discrete-event components

In this section, the method of checking consistency of DEC's with IAs is presented, which utilizes the environmental assumptions captured by these IAs.

**Derived interface automata** Before presenting the method, we need to have two auxiliary definitions – mirrors and input-universal RTS's of interface automata. The mirror of an IA  $A$  is built to represent all helpful environments with which  $A$  can be composed. A helpful environment of  $A$  is one that can always provide inputs expected by  $A$  and accept outputs generated by  $A$ . Also, any helpful environment of  $A$  should be an implementation of the mirror. In addition, we make explicit the environmental assumptions of IAs by building their input-universal RTS's, where a refused event will now be accepted but lead to an error state.

**Definition 8.** Given an IA  $A$ , the mirror of  $A$  is an IA  $M = (s_A^0, S_A, \Sigma_M, \rightarrow_A)$  with  $\Sigma_M^I = \Sigma_A^O$  and  $\Sigma_M^O = \Sigma_A^I$ ; The input-universal RTS of  $A$  is a deterministic RTS  $U = (s_A^0, S_A \cup \{\perp\}, \Sigma_A, \rightarrow_U)$ , where

$$\rightarrow_U = \rightarrow_A \cup \{(\perp, f, \perp) \mid f \in \Sigma_A^I\} \cup \{(s, f, \perp) \mid s \in S_A, f \notin \text{en}_A^I(s)\}.$$

Basically, the mirror of  $A$  has the input and output events of  $A$  interchanged. Hence  $\text{en}_M^I(s) = \text{en}_A^O(s)$  and  $\text{en}_M^O(s) = \text{en}_A^I(s)$  hold for all  $s \in S_A$ . The input-universal RTS of  $A$  is constructed by adding a transition outgoing from a state  $s \in S_A$  to a single error state  $\perp \notin S_A$  for all refused input events at  $s$ . As an example, the input-universal RTS of figure 2(a) is shown in figure 2(b), where the white triangle “ $\triangle$ ” represents the error state and “ $*$ ” matches any of input events of the RTS.

**Consistency checking** In the following, a two-step method of consistency checking is presented. Firstly, the input-universal RTS of the mirror of an IA is constructed. Next the product of the component and the RTS is built. The consistency is then determined by checking in the product for the absence of error and illegal deadlock states and the possibility of continuing interactions. This is justified by theorem 1 (below).

**Definition 9.** Consider a DEC  $C$  and an input-universal RTS  $U$  such that  $\Sigma_U^O \subseteq \alpha_C^I$ . The product of  $C$  and  $U$  is a RTS  $L_\otimes = (s_\otimes^0, S_\otimes, \Sigma_\otimes, \rightarrow_\otimes)$ , where:

- $s_\otimes^0 = \langle s_C^0, s_U^0 \rangle$  and  $S_\otimes \subseteq S_C \times S_U$  is the smallest set such that  $s_\otimes^0 \in S_\otimes$  and  $\forall s \in S_\otimes, s \xrightarrow{e}_\otimes s'$  implies  $s' \in S_\otimes$ .
- $\Sigma_\otimes^I = \Sigma_\otimes^O = \emptyset$ , and  $\Sigma_\otimes^H = \Sigma_C^{ctl} \cup (\Sigma_U^O \times \mathcal{U}^{val})$ ;
- $\rightarrow_\otimes = \{ \langle (q, s), e, \langle q', s' \rangle \rangle \mid e \in \Sigma_C^{ctl} \setminus (\Sigma_U^I \times \mathcal{U}^{val}), q \xrightarrow{e}_C q' \}$   
 $\cup \{ \langle (q, s), \langle f, v \rangle, \langle q', s' \rangle \rangle \mid \langle f, v \rangle \in \Sigma_U \times \mathcal{U}^{val}, q \xrightarrow{\langle f, v \rangle}_C q' \wedge s \xrightarrow{f}_U s' \}$

**Theorem 1.** Consider a DEC  $C$  and an IA  $A$  such that  $\Sigma_A^I \subseteq \alpha_C^I$ . Let  $L_\otimes$  be the product of  $C$  and the input-universal RTS  $U$  of  $A$ 's mirror. Then  $C$  is consistent with  $A$  if  $\forall \langle q, s \rangle \in S_\otimes$ , the following conditions hold:

1.  $s$  is not an error state, i.e.  $s \neq \perp$ ;
2. If  $\langle q, s \rangle$  is a terminal state, then  $s$  is a terminal state of  $A$ .
3. If  $\langle q, s \rangle$  is not a terminal state, then  $\exists s' \neq s, \langle q', s' \rangle \in S_\otimes$  such that  $\langle q', s' \rangle$  is reachable from  $\langle q, s \rangle$  in  $L_\otimes$ .

*Proof.* Let  $\hat{C}$  represent  $\hat{C}(\Sigma_A^O)$ ,  $\phi$  be a relation  $\{ \langle q, s \rangle \in S_\otimes \mid q \in S_{\hat{C}}, s \in S_A \}$ , we prove  $\phi$  is a simulation relation between  $\hat{C}$  and  $A$  by induction. First,  $(s_C^0, s_A^0) \in \phi$ . Next, suppose  $(q, s) \in \phi$ ,

1. If  $\text{en}_A^O(s) \neq \emptyset$ ,  $s$  is not a terminal state in  $A$ . Due to condition 2,  $\langle q, s \rangle$  is not a terminal state in  $L_\otimes$  either. Because mixed states are assumed to be absent in  $A$ ,  $\text{en}_A^I(s) = \text{en}_U^O(s) = \emptyset$ . Because of condition 3,  $\exists \langle q'', s \rangle \xrightarrow{\langle f, v \rangle}_\otimes \langle q', s' \rangle$  such that  $q \xrightarrow{f}_C q''$  and  $f \in \Sigma_U^I$  (note that  $\Sigma_U^I = \Sigma_A^O$ ). From definition 9,  $\langle f, v \rangle \in \text{en}_C^O(q)$  holds, i.e.  $\text{en}_C^O(q) \neq \emptyset$ .
2. For  $e \in \Sigma_C^{ctl} \setminus (\Sigma_U^I \times \mathcal{U}^{val})$ , if  $q \xrightarrow{e}_C q'$ , then  $q \xrightarrow{e}_\otimes q'$ . Hence  $(q', s) \in \phi$ ;
3. For  $\langle f, v \rangle \in \text{en}_C^O(q) \cup (\text{en}_A^I(s) \times \mathcal{U}^{val})$ , if  $q \xrightarrow{\langle f, v \rangle}_C q'$ , then  $q \xrightarrow{\langle f, v \rangle}_\otimes q'$ .  $\exists s' \in S_U$ ,  $s \xrightarrow{f}_U s'$  holds for  $f \in \text{en}_A^I(s)$ . It also holds for  $\langle f, v \rangle \in \text{en}_C^O(q)$  since  $\text{en}_C^O(q) \subseteq \Sigma_A^O \times \mathcal{U}^{val}$  and  $U$  is input-universal w.r.t.  $\Sigma_A^O$ . From definition 9, we can get  $\langle q', s' \rangle \in S_\otimes$ . Due to condition 1,  $(q', s') \in \phi$  holds.

Therefore,  $\phi$  is a simulation relation between  $\hat{C}$  and  $A$ . From definition 7,  $C$  is consistent with  $A$ .  $\square$

In the theorem, condition 1 indicates the input and output duality between  $C$  and  $A$ . Condition 2 requires the absence of illegal deadlock states in the product. Finally, condition 3 states a requirement on the reactive nature of  $C$ , that is,  $C$  should be active in communication.

Now we are able to check the consistency of the store component of figure 1 with the store automaton of figure 2(a). We calculate the product of the component model and the input-universal RTS of the automaton's mirror and check it against the above conditions. If these conditions are satisfied, then theorem 1 allows us to conclude that the store component is consistent with the store automaton. At the time of writing, this algorithm has been implemented in the context of Moses.

## 4 Modular analysis of component networks

### 4.1 Dataflow process networks

There are many kinds of process networks [10–12, 15]—they differ, e.g., in their model of communication (explicit FIFO buffers between processes vs synchronous communication), or in their model of execution (as an interleaving of atomic and non-blocking firings of processes vs a continuous and possibly blocking thread-like execution of each process in parallel with all other processes).

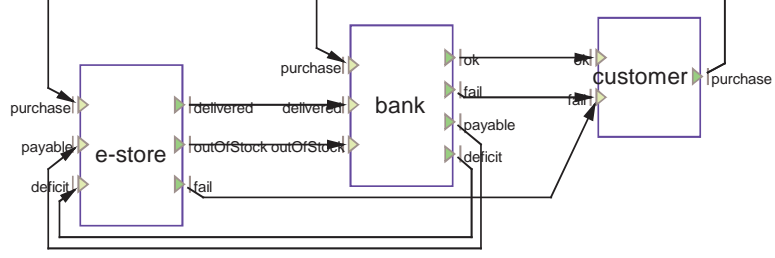
In this paper we consider the form proposed in [15]. Basically, a process network consists of a collection of concurrently executing processes with ports and a set of channels connecting the output and input ports of these processes. Often, the channels represent FIFO buffers between processes, but we consider that the buffers are encapsulated in their destination processes and the channels represent only the causality of data flow between processes. Due to the localization of buffers, the semantic definition of process networks is simplified and thus facilitates modular analysis. Furthermore, it also gives us the flexibility to model a variety of buffers thanks to the diversity of component modelling formalisms.

**Definition 10.** A dataflow process network (DPN) is defined by  $D = (P, R)$ , where  $P \subset \mathcal{U}^{dec}$  is a set of processes and  $R \subseteq \bigcup_{p \in P} \alpha_p^O \times \bigcup_{p \in P} \alpha_p^I$  is a set of connections relating the output ports to the input ports of the processes, such that <sup>5</sup>

- $(o, i) \in R$  implies  $\rho(o) \neq \rho(i)$ ;
- $(o, i), (o, i') \in R \wedge i \neq i'$  implies  $\rho(i) \neq \rho(i')$ ;

where  $\rho(f) = p$  if  $p \in P$  and  $f \in \alpha_p^I \cup \alpha_p^O$ .

<sup>5</sup> The function  $\rho(f)$  returns the process associated with port  $f$ . Here, we exclude the situations where  $R$  connects an output port and an input port of one process and where more than one connection originating from one output port ends at two or more input ports of a process. This is because these introduce true concurrency at component boundaries, which in turn contradicts the interleaving semantics of interface automata. This will be addressed in future work.



**Fig. 3.** An online purchase DPN

A DPN can be depicted as a directed graph. At this level of abstraction, each node represents a process, each triangle associated with a node represents an input/output port of the node, and each edge represents a connection between ports. Figure 3 shows an example: an online purchase DPN. When a customer sends a purchase request, the request goes simultaneously to the store and the bank. These then collaborate to process the request and finally report back to the customer whether the purchase succeeds or fails (detail will be given later).

A variety of communication structures are supported by DPNs where data can be relayed, duplicated, and merged among processes. For example, two connections starting from the port “purchase” of the customer and the connections ending at the port “fail” of the customer in figure 3 demonstrate the last two situations, respectively. Furthermore, disconnected input and output ports of processes in a DPN are allowed. A disconnected input port will receive no data, while a disconnected output port will discard all data sent to it. We further define the sets of *connected input and output ports* of a process  $p \in P$  in a DPN  $D$  as  $\bar{\alpha}_p^I = \{i \in \alpha_p^I \mid (o, i) \in R\}$  and  $\bar{\alpha}_p^O = \{o \in \alpha_p^O \mid (o, i) \in R\}$ , respectively.

**Definition 11.** Consider a DPN  $D = (P, R)$ . Let  $R^\Sigma = \{(\langle o, v \rangle, \langle i, v \rangle) \mid (o, i) \in R, v \in U^{val}\}$  be a causality relation between output and input events. Then the product of  $D$  is defined as a RTS  $L_D = (s^0, S, \Sigma, \rightarrow_D)$ , where

- $s^0 = \prod_{p \in P} s_p^0$  and  $S \subseteq \prod_{p \in P} S_p$ . We let projections  $\pi_p: S \rightarrow S_p$  and let  $s_p = \pi_p(s)$  and  $s'_p = \pi_p(s')$  for  $p \in P, s, s' \in S$ ;
- $\Sigma^I = \Sigma^O = \emptyset$  and  $\Sigma^H = \bigcup_{p \in P} \Sigma_p^{ctl}$ ;
- $\rightarrow_D = \{(s, e, s') \mid e \in \Sigma_p^{ctl}, s_p \xrightarrow{e} s'_p \wedge \forall g \in P, g \neq p \wedge s'_g = \delta(s_g, p, e)\}$

$$\text{where } \delta(s_g, p, e) = \begin{cases} q & \text{if } e \in \Sigma_p^O \wedge \exists e' \in \Sigma_g^I, q \in S_g, (e, e') \in R^\Sigma \wedge s_g \xrightarrow{e'} q. \\ s_g & \text{otherwise} \end{cases}$$

The product of a DPN captures the semantics of the DPN. According to the definition, a state of a DPN is a vector of states of all its processes, and a DPN transits between states by simply executing one of its processes and directing data flow, if any, according to the connections  $R$ . A transition of the DPN is either an internal transition or an output transition of a process. The latter may involve synchronous execution of multiple input transitions of other processes, depending on  $R$ .

In order to facilitate further analysis, we define projections of traces of DPNs, which relates the behaviour of a DPN to that of its processes.

**Definition 12.** Given a DPN  $D = (P, R)$  and a trace  $\sigma$  of  $L_D$  from  $s_D^0$ , the trace projection of  $\sigma$  on  $p \in P$  is a trace of  $p$  from  $s_p^0$ , denoted as  $\pi_p(\sigma)$ , obtained by first removing from  $\sigma$  all events not in  $\Sigma_p^{ctl} \cup X$  and then renaming all events  $x \in X$  to  $y \in \Sigma_p^I$  s.t.  $(x, y) \in R^\Sigma$ , where  $X = \{x \mid \exists y \in \Sigma_p^I, (x, y) \in R^\Sigma\}$ .

## 4.2 Interface automaton networks

We define the composition of the IAs by interface automaton networks as for DECs.

**Definition 13.** An interface automaton network (IAN) is defined as  $N = (W, R)$ , where  $W \subset \mathcal{U}^{ia}$  and  $R \subseteq \bigcup_{a \in W} \Sigma_a^O \times \bigcup_{a \in W} \Sigma_a^I$  is a causality relation between the output and input events of the IAs, such that<sup>6</sup>:

- $(o, i) \in R$  implies  $\rho(o) \neq \rho(i)$ ;
- $(o, i), (o, i') \in R \wedge i \neq i'$  implies  $\rho(i) \neq \rho(i')$ ;

where  $\rho(f) = a$  if  $a \in W$  and  $f \in \Sigma_a$ .

The semantics of IANs is captured by their products defined below.

**Definition 14.** Consider an IAN  $N = (W, R)$ . Let  $B$  be the set of input-universal RTSs of all IAs in  $W$ . Then the product of  $N$  is defined as a RTS  $L_N = (s^0, S, \Sigma, \rightarrow_N)$ , where:

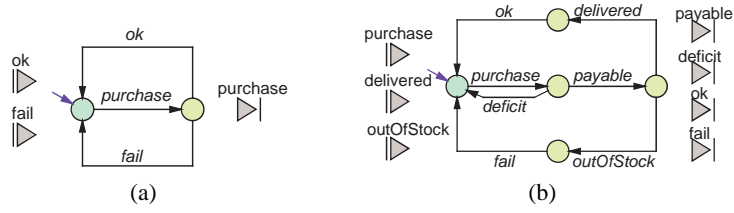
- $s^0 = \prod_{b \in B} s_b^0$  and  $S \subseteq \prod_{b \in B} S_b$  is the smallest set such that  $s^0 \in S$  and  $\forall s \in S, s \xrightarrow{f}_N s'$  implies  $s' \in S$ . We let projections  $\pi_b: S \rightarrow S_b$  and let  $\mathbf{s}_b = \pi_b(\mathbf{s})$  and  $\mathbf{s}'_b = \pi_b(\mathbf{s}')$  for  $b \in B, \mathbf{s}, \mathbf{s}' \in S$ ;
- $\Sigma^I = \Sigma^O = \emptyset$ , and  $\Sigma^H = \bigcup_{b \in B} \Sigma_b^O$ ;
- $\rightarrow_N = \{(s, f, s') \mid f \in \Sigma_b^O, \mathbf{s}_b \xrightarrow{f}_b \mathbf{s}'_b \wedge \forall h \in B, h \neq b \wedge \mathbf{s}'_h = \delta(\mathbf{s}_h, f)\}$ ,  
where  $\delta(\mathbf{s}_h, f) = \begin{cases} q & \text{if } \exists i \in \Sigma_h^I, q \in S_h, (f, i) \in R \wedge \mathbf{s}_h \xrightarrow{i}_h q. \\ \mathbf{s}_h & \text{otherwise} \end{cases}$ .

As an example, suppose that we have an IAN where  $W$  consists of the interface automata of figures 2(a), 4(a) and 4(b), and  $R$  defines their composition as shown in figure 3. Then the product of the IAN is shown in figure 5.

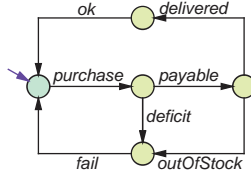
**Definition 15.** Consider an IAN  $N = (W, R)$ . Let  $L_N$  be the product of  $N$ . Then  $N$  is consistent if no error or deadlock states are reachable in  $L_N$ , i.e.  $\forall s \in S_N$ , the following conditions hold:

1.  $\forall a \in W, \pi_a(\mathbf{s}) \neq \perp$ ;
2. If  $\mathbf{s}$  is a terminal state of  $L_N$ , then  $\forall a \in W, \text{en}_a^I(\pi_a(\mathbf{s})) = \emptyset$ .

<sup>6</sup> These well-formedness rules are introduced for the same reason as in definition 10.



**Fig. 4.** The customer automaton (a) and the bank automaton (b)



**Fig. 5.** The product of the example IAN

Basically, the consistency of IANs ensures the absence of mismatches of environmental assumptions and output guarantees of processes and also the freedom of deadlock at a high level of abstraction. For example, as the product state space shown in figure 5 contains no error or deadlock state, the IAN is consistent.

Since IAs can specify the interface behaviour of DECs, IANs can capture the interaction behaviour of DECs in DPNs. Hence, as we shall see, the consistency of IANs can serve as the basis of analysis of DPNs. Also, it is cheaper to determine the consistency of IANs, because IANs generally have smaller state space than DPNs.

### 4.3 Properties of dataflow process networks

In this section, we define the properties of DPNs such as safety and deadlock freedom. The safety considered here refers to the fact that the environmental assumptions made by processes are respected in executions of DPNs. More specifically, no unexpected reception of data at any input port ever occurs. Additionally, a DPN is said to be deadlocked if it reaches a state where no process can make any progress, generally because each is blocked waiting for an input from others, while the event cannot occur. Deadlock freedom refers to the ability of DPNs to make progress or perform computations.

**Definition 16.** A dataflow process network  $D$  is sketched by a total function  $\mathcal{A}: P \rightarrow \mathcal{U}^{ia}$  if  $\forall (p, a) \in \mathcal{A}, \Sigma_a^I = \bar{\alpha}_p^I, \Sigma_a^O = \bar{\alpha}_p^O$  and  $p$  is consistent with  $a$ , where  $\bar{\alpha}_p$  denotes the connected ports of  $p$  in  $\hat{D}$  as defined in section 4.1.

Consider a DPN  $D$  sketched by a total function  $\mathcal{A}$ . Let  $L_D$  be the product of  $D$ ,  $\sigma$  be a trace of  $L_D$  from  $s_D^0$ ,  $p \in P$ ,  $a = \mathcal{A}(p)$ ,  $\hat{p}$  represent  $\hat{p}(\bar{\alpha}_p^O)$ ,  $\xi_p = \pi_p(\sigma) \upharpoonright \Sigma_p^{obs}$  be the observable sequence of the trace projection of  $\sigma$  on  $p$ , and  $\xi_a$  be the sequence of ports involved in  $\xi_p$  (Note  $|\xi_p| = |\xi_a|$ ). Then we can formulate properties of  $D$  in definitions 17 and 18 below.

**Definition 17.** A trace  $\sigma$  is free of unexpected reception in  $D$  if  $\xi_a$  is a trace of  $a$  from  $s_a^0$  for all  $p \in P$ .  $D$  is free of unexpected reception if all traces of  $L_D$  from  $s_D^0$  are free of unexpected reception.

**Lemma 1.** Consider a trace  $\sigma$  which is free of unexpected reception. Let  $\mathbf{q} \in S_D$  be a reachable state via  $\sigma$  in  $L_D$  and  $\mathbf{s}_a$  be a reachable state via  $\xi_a$  in  $a$ , then (1)  $\mathbf{s}_a$  is the only state reachable via  $\xi_a$  in  $a$  and (2)  $\pi_p(\mathbf{q}) \preceq \mathbf{s}_a$ .

*Proof.* (1) holds because  $\Sigma_a^H = \emptyset$  and  $a$  is assumed to be deterministic. Also, because every event  $f$  in  $\xi_a$  corresponds an event  $\langle f, v \rangle$  in  $\xi_p$ , (2) holds from definition 7.  $\square$

**Definition 18.** Consider a DPN  $D$  which is free of unexpected reception. Let  $\mathbf{q} \in S_D$  be a reachable state via  $\sigma$  in  $L_D$  and  $\mathbf{s}_a$  be the reachable state via  $\xi_a$  in  $a$  for all  $p \in P$ .  $\mathbf{q}$  is a deadlock state if it is a terminal state in  $L_D$  and  $\exists p \in P, \text{en}_a^I(\mathbf{s}_a) \neq \emptyset$ .  $D$  is free of deadlock if no deadlock state is reachable via any trace from  $s_D^0$ .

#### 4.4 Property deduction

**Theorem 2.** A DPN  $D = (P, R)$  is free of unexpected reception and deadlock if there exist both a total function  $\mathcal{A}: P \rightarrow \mathcal{U}^{ia}$  s.t.  $D$  is sketched by  $\mathcal{A}$  and also a consistent IAN  $N = (W, R)$ , where  $W = \{\mathcal{A}(p) \mid p \in P\}$ .

*Proof.* We prove this theorem by induction over the length of any trace  $\sigma$  from  $s_D^0$ . Let  $\mathbf{q} \in S_D$  be a reachable state via  $\sigma$ ,  $p \in P$ ,  $a = \mathcal{A}(p)$ ,  $\hat{p}$  represent  $\hat{p}(\bar{\alpha}_p^O)$ ,  $\xi_p = \pi_p(\sigma) \upharpoonright \Sigma_p^{obs}$  and  $\xi_a$  be the sequence of ports involved in  $\xi_p$ . At each step, we prove that (1)  $\sigma$  is free of unexpected reception; (2)  $\exists \mathbf{s} \in S_N, \forall a, \pi_a(\mathbf{s})$  is the reachable state via  $\xi_a$  in  $a$ ; and (3)  $\mathbf{q}$  is not a deadlock state.

1. If  $\sigma = \lambda$ , then  $\mathbf{q} = s_D^0$ . Clearly, (1) holds. Let  $\mathbf{s} = s_N^0$ , then (2) holds. Suppose that  $\mathbf{q}$  is a terminal state in  $L_N$ , i.e.  $\forall p, \nexists e \in \Sigma_p^{ctl}, (\pi_p(\mathbf{q}), e, q) \in \rightarrow_p$ . Hence,  $\forall p, \text{en}_p^O(\pi_p(\mathbf{q})) = \emptyset$ . Because  $\pi_p(\mathbf{q}) \preceq \pi_a(\mathbf{s}), \forall a \in W, \text{en}_a^O(\pi_a(\mathbf{s})) = \emptyset$  and thus  $\mathbf{s}$  is a terminal state. Because  $\mathbf{s} \in S_N$  and thus  $\mathbf{s}$  is not a deadlock state, we have  $\forall a, \text{en}_a^I(\pi_a(\mathbf{s})) = \emptyset$ . Therefore, (3) holds.
2. Suppose  $\sigma = e_1 e_2 \dots e_m$  s.t. (1-3) hold on  $\sigma$ . Given a trace  $\sigma' = \sigma \cdot e$ , we shall prove (1-3) hold on  $\sigma'$ . Let  $\mathbf{s} \in S_N$  be the state satisfying (2),  $\mathbf{q}_p = \pi_p(\mathbf{q})$  and  $\mathbf{s}_a = \pi_a(\mathbf{s})$  for all  $p \in P$ , and  $\xi'_p$  and  $\xi'_a$  are defined over  $\sigma'$ . Then from lemma 1 we have  $\forall p, \mathbf{q}_p \preceq \mathbf{s}_a$ .
  - (a) if  $e \in \Sigma_p^{ctl} \setminus (\Sigma_a^O \times \mathcal{U}^{val})$ , then  $\xi'_a = \xi_a$  and (1) holds. Let  $\mathbf{s}' = \mathbf{s}$ , then  $\mathbf{s}' \in S_N$  and  $\mathbf{s}'$  is reachable via  $\sigma'$ . Thus (2) holds. Same as item 1, we can prove (3).
  - (b) if  $e \in \Sigma_a^O \times \mathcal{U}^{val}$  and  $\mathbf{q}_p \xrightarrow{e}_p \mathbf{q}'_p$ , let  $e = \langle f, v \rangle$ , then  $\exists \mathbf{s}'_a \in S_a, \mathbf{s}_a \xrightarrow{f}_a \mathbf{s}'_a \wedge \mathbf{q}'_p \preceq \mathbf{s}'_a$  (because  $\mathbf{q}_p \preceq \mathbf{s}_a$ ). Thus  $\xi'_a = \xi_a \cdot f$  is a trace of  $a$ . For  $g \in P \wedge g \neq p$ , we let  $h = \mathcal{A}(g)$ . Then,
    - i. if  $\exists \langle f', v \rangle \in \Sigma_g^I, (f, f') \in R$ , then  $\exists \mathbf{q}'_g \in S_g, \mathbf{q}_g \xrightarrow{\langle f', v \rangle}_g \mathbf{q}'_g$ . Since no error state exists in  $L_N$  (def. 15),  $f' \in \text{en}_h^I(\mathbf{s}_h)$  and thus  $\exists \mathbf{s}'_h \in S_h, \mathbf{s}_h \xrightarrow{f'}_h \mathbf{s}'_h$  such that  $\mathbf{q}'_g \preceq \mathbf{s}'_h$ . Hence  $\xi'_h = \xi_h \cdot \langle f', v \rangle$  is a trace of  $h$ .
    - ii. otherwise,  $\xi'_h = \xi_h$  is a trace of  $h$ .

Therefore, (1) holds on  $\sigma'$ . From def. 14,  $\exists s' \in S_N, \forall a, \pi_a(s') = s'_a$  and  $s'_a$  is reachable via  $\xi'_a$ . Hence, (2) holds on  $\sigma'$ . (3) can be proved on  $\sigma'$  as in item 1.

Therefore, the theorem holds.  $\square$

With this theorem, we can conclude the example process network of figure 3 is free of unexpected reception and deadlock, provided that the concrete component models of the bank and the customer are consistent with their corresponding interface automata, respectively.

In the context of Moses, we have also implemented the check for consistency of IANs. This, together with the check based on theorem 1, gives us the ability to analyse DPNs for properties such as freedom from deadlock and unexpected reception.

## 5 Conclusion

In this paper a modular analysis method for dataflow process networks has been presented, where interface automata are associated with processes (or components) to specify both their interface behaviour requirements and possible environmental assumptions. Based on the IAs, we deduce the properties of DPNs, such as freedom from unexpected reception and deadlock, by checking the consistency of components and of interface automaton networks. As these checks only need to handle smaller state spaces than the traditional single monolithic check, the state space explosion problem can be alleviated. At this stage, we have implemented the algorithms for checking these two kinds of consistency in the Moses tool suite, with the development of a visual notation for interface automata and tools for their composition and compatibility checking.

The introduced interface automata can specify the behaviour of components at a high level of abstraction and serve as the contracts between architecture designers and component developers. In this way, highly independent development of components and the communication structure among components is supported. Also, this acknowledges that a system is usually designed with assumptions made about the abstract behaviour of components and that components are designed assuming particular interaction patterns with their environment.

In addition, the proposed method simplifies substitutability checking between heterogeneous components using an intermediate interface automaton. That is to say, a component can be substituted by another component in a process network if they are both consistent with the same interface automaton. Hence, the evolution of systems is supported both at the abstract level by the substitutability of interface automata and also at the component level by the substitutability of components.

The research presented here is a step towards the automated consistency checking of heterogeneous systems where system components as well as system architectures are potentially expressed in different description languages. We are investigating the application of this method to architectural models described in other languages such as Petri Nets. Currently, the assumptions of components on data values are not captured in this method. A possible way to improve this is to enhance the formalism of interface automata to support data values on input and output events. Furthermore, true concurrency at component boundaries is not considered here and will be the subject of future work.

## References

1. M. Anlauff, P. Kutter, A. Pierantonio, and A. Sünbül. Using domain-specific languages for the realization of component composition. In *Proc. of the Fundamental Approaches to Software Engineering (FASE 2000)*, LNCS 1783. Springer.
2. F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, 2000.
3. L. de Alfaro and T. Henzinger. Interface automata. In *Proc. of the Foundation of Software Engineering (FSE 2001)*, pages 109–122. ACM Press.
4. R. Esser and J. Janneck. Moses - a tool suite for visual modelling of discrete-event systems. In *Symposium on Visual/Multimedia Approaches to Programming and Software Engineering*, 2001.
5. P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. In *Proc. of the Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029.
6. P. Inverardi, A. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Trans. on Software Engineering and Methodology*, 9(3):239–272, 2000.
7. J. Janneck and R. Esser. Higher-order Petri Net modeling—techniques and applications. In *Workshop on Softw. Eng. and Formal Methods of ICATPN 2002*.
8. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.
9. Y. Jin, R. Esser, and J. Janneck. Describing the syntax and semantics of UML statecharts in a heterogeneous modelling environment. In *Proc. of the Diagrammatic Representation and Inference (Diagrams 2002)*, LNAI 2317. Springer.
10. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 1974*, pages 471–475. North-Holland Publishing Co.
11. R. Karp and R. Miller. Properties of a model for parallel computations: determinacy, termination, queuing. *SIAM J. Appl. Math.*, 14:1390–1411, 1966.
12. E. Lee and T. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, 1995.
13. S. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Proc. of the Computer-Aided Verification (CAV 2002)*, LNCS 2404. Springer.
14. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, Advances in Petri Nets, LNCS 1491. Springer-Verlag, 1998.
15. D. Skillcorn. Stream languages and data-flow. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, 1991.
16. S. Uchitel and D. Yankelevich. Enhancing architectural mismatch detection with assumptions. In *Proc. of the Eng. of Computer Based Systems (ECBS 2000)*.
17. D. Yellin and R. Storm. Protocol specifications and component adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.