

# Lightweight Consistency Analysis of Dataflow Process Networks

Yan Jin

Robert Esser

Charles Lakos

Department of Computer Science, University of Adelaide, Adelaide, SA 5005, Australia  
Email: {yan, esser, charles}@cs.adelaide.edu.au

## Abstract

Process networks are a popular modelling technique for distributed computing and signal processing applications. The ability to support various parallelism or communication patterns also makes them suitable for modelling multi-processor architectures. At the architecture description level, the language provides the flexibility to model actual processes using various formalisms. This is especially important when the systems are comprised of parts with distinct characteristics, *e.g.* control-based or dataflow-oriented. However, this heterogeneity of processes poses a challenge for the consistency analysis of process networks. This research proposes a lightweight method for analyzing the consistency of such networks. The method employs interface automata as a bridge between the architectural model and heterogeneous components representing concrete models of processes. Utilising interface automata, consistency is determined by a series of small tasks at both the architecture level and the component level. This separation of concerns simplifies the handling of heterogeneous components and alleviates the potential state space explosion problem when analyzing large systems.

**Keywords:** component-based systems, consistency analysis, dataflow process networks

## 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 quality improvement 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. Even with additional informal descriptions, such specifications are not adequate for designing reliable and evolving software systems. Instead, what is needed are more rigorous specifications which capture interface behaviours of components, including what services a component provides, how it is properly deployed, and the dependencies between its inputs and outputs. Naturally these specifications must not disclose implementation details of their associated 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

(Anlauff, Kutter, Pierantonio & Sünbül 2000). Instead, it is preferable to provide flexible connectors so that component-based systems can be constructed using various design strategies (Bachman, Bass, Buhman, Comella-Dorda, Long, Robert, Seacord & Wallnau 2000), where suitable architectural styles, *e.g.* pipe-and-filter and client/server architectures, can be employed. These styles describe the types of components and the allowable interaction patterns at a high level of abstraction. The major challenge is then to ensure that the resulting systems are consistent (namely, all components are deployed properly in the design) and that these systems meet global functional and nonfunctional requirements such as structural invariants, reliability and security. This problem is even more important when designing hierarchical systems comprising of components with internal architectures, where different styles may be applied at different levels.

This paper represents a step towards the component-based development and modular analysis of dataflow process networks. Here, components (or processes) communicate through their input and output ports and the interconnection of components specifies a causality relation between data flow through input and output ports of the components.

Usually, the consistency of such a network is described by some safety properties over communications among components. To prove consistency, model checkers have to be employed which construct the whole state space of the network and check it against these properties. However, this approach suffers from the state space explosion problem as the state space of the network grows exponentially with the number of its components (or processes).

In order to avoid this problem, the composition of components is not analysed directly in this approach. Instead, interface automata (de Alfaro & Henzinger 2001) are associated with components, specifying the external interaction protocol of the components at a high level of abstraction. The protocols can capture not only how components react to their inputs but also the assumptions of components on when or what inputs are expected. Acting as the contracts between components, these protocols are the key for enabling modular analysis of process networks. Also, with the ability of interface automata to capture the input assumptions of components, we obviate the need for explicitly specifying the consistency properties of process networks (de Alfaro & Henzinger 2001).

The consistency check is now a two-stage process. Firstly, each component in a process network is checked for consistency with its corresponding interface automaton, namely, it communicates with the environment in a way that conforms to the external interaction protocol described by the automaton. More specifically, all the outputs which can be gener-

ated by the component can also be generated by the automaton, provided that the input assumptions captured by the automaton are satisfied by the context of the component. In this step, the same two-stage process can be recursively applied for components with internal structure defined as process networks.

Secondly, the appropriateness of the interconnections defined by a process network is checked using interface automata. This involves the construction and consistency checking of an interface automaton network, which consists of the associated automata and shares the same interconnections with the process network. The consistency of the automaton network implies that data flow between components can be directed in a way which respects the input assumptions of the components, and can in turn justify the consistency of the process network.

As interface automata abstract away the implementation details of components, the consistency checking of interface automaton networks is much cheaper than that for dataflow process networks. Also, the interleaving of internal behaviour of components does not need to be explored. Therefore, this divide-and-conquer approach can help alleviate the state space explosion problem.

This research is motivated by previous work on the Moses tool suite (Esser & Janneck 2001). Moses presents an additional challenge for component-based software development in that it supports the modelling of heterogeneous discrete-event systems, where components can be described by different formalisms (Esser & Janneck 2001, Janneck & Esser 2002, Jin, Esser & Janneck 2002), for example, process networks, Petri Nets, Statecharts, etc. The proposed approach can also largely simplify the handling of heterogeneous components when checking the consistency of such heterogeneous component-based systems.

This paper is structured as follows. In section 2, our approach is compared with the related work. In section 3, we define the underlying concepts such as discrete-event components and interface automata, and present our method for checking the consistency of the former with respect to the latter. In section 4, dataflow process networks are defined and the method of determining their consistency is presented. Finally, we conclude this paper in section 5.

## 2 Related work

The term *consistency* has been used by the research community with different meanings in different contexts, *e.g.* the cache consistency of cache coherent protocols (Delzanno 2000), the consistency between multiple viewpoints of systems (Fradet, Métayer & Périn 1999, Sunetnanta & Finkelsteing 2001, Nuseibeh, Easterbrook & Russo 2001), and the compatibility and substitutability of components in component-based systems (van der Aalst, van Hee & van der Toorn 2002, Yellin & Storm 1997, Inverardi, Wolf & Yankelevich 2000, Uchitel & Yankelevich 2000). Our focus is on component-based systems.

First of all, our interpretation of consistency is inspired by (van der Aalst et al. 2002). There, components and their interface behaviour requirements are described by a variant of classical Petri Nets, called *component nets (C-nets)*. The consistency between components and their substitutability are defined in terms of *projection inheritance*. More specifically, a component model is said to be a subclass of its behaviour requirement model under projection inheritance if and only if the former is a *branching bisimulation* (van Glabbeek & Weijland 1996) of the latter after hiding all additional methods. This relation-

ship does not allow components to provide more services than its requirement model, which may limit the reusability of components. In our approach, alternating simulation (de Alfaro & Henzinger 2001) is employed to define consistency, which allows implementations to handle more legal inputs (or service requests) than specified by its requirement. Furthermore, the components in (van der Aalst et al. 2002) are different in nature from ours. Their components are not input-universal (*i.e.* an input can be refused) and the decision when to fetch an input is controlled by the component itself. In our approach components are required to be input-universal (*i.e.* an input is never refused) and hence the environment controls when a component receives an input. In addition, in our approach the substitutability of heterogeneous components can be checked with the aid of interface automata.

Interface automata were first introduced in (de Alfaro & Henzinger 2001). The authors established simple but 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 environment by assuming that it is always helpful, only supplying inputs expected by an automaton and always accepting an output produced by the automaton. This optimistic view allows more possible implementations than a pessimistic approach where the environment can behave as it pleases. Furthermore, an algorithm for refinement checking between two interface automata was given, which first computes the Cartesian product of the state spaces of the automata and then recursively removes from the product states that do not meet the refinement conditions until a fix-point is reached. In our approach, alternating simulation is adapted to define the consistency of discrete-event components with interface automata, which takes into account data values in components. As components generally have infinite state spaces, the same checking algorithm cannot apply. Instead, we propose a different method where the mirror of an interface automaton is constructed that represents all helpful environments of a component to be checked. The consistency is then determined by the absence of error states in the product of the component and the mirror. Our method is simpler as it does not require the construction of the Cartesian product of the state spaces of the specification and the implementation — only the reachable states in the product needs to be constructed. Also, in contrast to their simple composition scheme of interface automata, we allow interface automata to be composed in many more ways reflecting how process networks can be constructed.

The construction of mirrors in our approach is inspired by (Rajamani & Rehof 2002), where mirrors are built to check the conformance (or refinement) relationship between models of asynchronous message passing software written in the Calculus of Communicating Systems (CCS) (Milner 1989). There, mirrors are used to represent all possible environments of component models. Similar to other pessimistic approaches, the authors consider that the conformance must hold under an arbitrary environment. This restriction on implementation models is stronger than in our approach. Furthermore, this approach requires that not only the specification but also the implementation models have no mixed states (where both input and output transitions originate), while in our approach this is only required of the specification.

There are some other approaches which also utilize the assumptions that components can make on the environment for the verification of component-based

systems. In (Inverardi et al. 2000), 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. In contrast, our method is complete and allows flexible communication patterns including many-to-one and one-to-many synchronization.

The approach in (Uchitel & Yankelevich 2000) 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

Components are a critical part of a component-based system design. Systems, especially embedded systems, are often composed of components with distinct characteristics, *e.g.* state-based or dataflow-oriented. Here it is preferable to design each component using the most suitable description language. In order to support this kind of heterogeneous component-based modelling, a formal model of discrete-event components (Janneck 2000) is defined in section 3.1. This model is sufficiently general so that the semantics of components written in various formalisms can be easily defined.

In order to facilitate stand-alone development and analysis of components, we employ interface automata to specify the desirable interaction protocols of components with the environment. The protocols cover both the output requirements of components and also the input assumptions that components can make. In section 3.2, the use of interface automata in this approach is given. In the next section, the consistency of components with respect to interface automata is defined. Following two auxiliary definitions of derived interface automata in section 3.4, a method of checking the consistency is proposed in section 3.5.

#### 3.1 Discrete-event components

In Moses, a component can be represented as a discrete-event component that consumes data streams fed to its input ports, and produces data streams from its output ports. The input and output ports form a component's view of the rest of the system and decouple the outside world and the component. This separation allows the behaviour of the component to be described independently from the component's 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 finite universal set  $\mathcal{U}^{val}$  of values of data flowing through the ports.

**Definition 1.** A *discrete-event component (DEC)*<sup>1</sup> is defined by  $C = (s^0, S, \Sigma, \rightarrow)$ , where:

<sup>1</sup>The definition of DEC's in (Janneck 2000) allows a component to have more than one output when firing an output transition. This introduces true concurrency at component boundaries. This can be handled by employing true concurrency semantics such as event structures (Winskel 1987) and Mazurkiewicz traces (Mazurkiewicz 1989). However, we restrict our attention to the interleaving semantics of components. That is, we require that firing an output transition can generate only one output. The handling of true concurrency is left for future work.

- $S$  is a set of states and  $s^0 \in S$  is the initial state;
- $\Sigma \subset \mathcal{U}^{port}$  is a finite set of ports, consisting of two disjoint sets of input ports  $\Sigma^I$  and output ports  $\Sigma^O$ . We let a set of labels  $L = \{\tau\} \cup (\Sigma \times \mathcal{U}^{val})$ , where  $\tau \notin \mathcal{U}^{port} \times \mathcal{U}^{val}$  is used to label transitions with no external effects;
- $\rightarrow \subseteq S \times L \times S$  is a set of transitions such that an input-universal requirement holds, that is,  $\forall s \in S, e \in \Sigma^I \times \mathcal{U}^{val}, \exists (s, e, s') \in \rightarrow$ .

A component  $C$  is called *closed* if  $\Sigma^I = \Sigma^O = \emptyset$ ; otherwise, it is said to be *open*. An *execution fragment* of  $C$  is a finite alternating sequence of states and labels  $s_1, l_2, s_2, \dots, l_k, s_k$  such that  $\forall j: 2 \leq j \leq k, s_1, s_j \in S, l_j \in L, (s_{j-1}, l_j, s_j) \in \rightarrow$ . For the execution fragment,  $s_k$  is called *reachable* from  $s_1$ . Also,  $s_k$  is said to be *internally reachable* from  $s_1$  if  $\forall j: 2 \leq j \leq k, l_j = \tau$ , and to be *reachable in  $C$*  if  $s_1$  is the initial state  $s^0$ .

In the sequel, we simply write  $s \xrightarrow{l} s'$  to denote  $(s, l, s') \in \rightarrow$ . We also write  $s \Longrightarrow s'$  to denote the fact that either  $s' = s$  or  $s'$  is internally reachable from  $s$ , and  $s \xrightarrow{l} s'$  to denote  $l \neq \tau \wedge \exists s'' \in S, s \Longrightarrow s'' \wedge s'' \xrightarrow{l} s'$ . In addition, we assume a universal set  $\mathcal{U}^{dec}$  of discrete-event components, and that ports of all components are distinct.

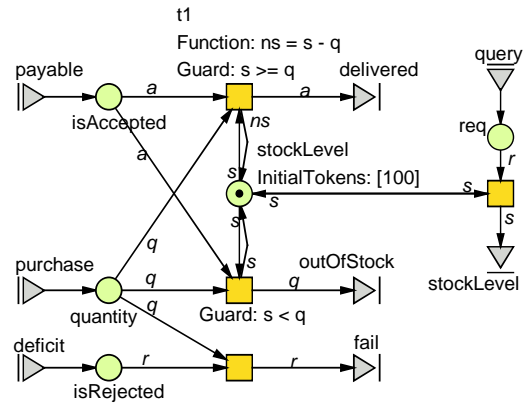


Figure 1: An online store “e-store”

Figure 1 gives an example DEC depicted in a variant of Petri Nets (Janneck 2000), where triangles represent the input and 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.* “t1”) becomes enabled once all its predecessor places have enough tokens and its guard evaluates to true. As is the case for other high-level Petri nets (*e.g.* (Jensen 1997)), this binds the tokens to the variable names (*e.g.* “s” and “q”) 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>2</sup>. For a Petri Net component, we require input ports to be connected to places and output ports to transitions to ensure

<sup>2</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.

that any component output can be meaningfully connected to any component input. Taking the interleaving semantics of Petri Net components (Janneck & Esser 2002), the interpretation of such components in terms of discrete-event components is straightforward and we omit this for the sake of brevity. See (Reisig & Rozenberg 1998) for the basic concepts of Petri Nets, and (Esser & Janneck 2001, Janneck & Esser 2002) for more descriptions of the Moses approach to compositional Petri Nets.

The example shown in figure 1 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. 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 to be subtracted by its ordered quantity.

A Moses component is input-universal, *i.e.* it never 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 (Janneck & Esser 2002) may have multiple places acting as explicit buffers, while a UML Statechart component (Jin et al. 2002) has only one implicit buffer for all input ports. These built-in component buffers ensure the acceptance of inputs.

### 3.2 Interface automata

Usually, a component is designed under some assumptions about the environment depicting how the component can be properly deployed, for example, interaction protocols. The assumptions are useful for analyzing the behaviour of the component, especially when the component is independently developed and analysed. However, from section 3.1 a component is required to be input-universal, namely, it cannot constrain the environment as to when or what kind of input to provide. Therefore, these assumptions cannot be captured by component models themselves. To solve this problem, we employ interface automata introduced in (de Alfaro & Henzinger 2001) to specify desirable interaction protocols of components, which cover both the input assumptions and the output behaviour of components.

**Definition 2.** An *interface automaton (IA)* is defined as  $A = (s^0, S, \Sigma, \rightarrow_A)$ , where:

- $S$  is a finite set of states and  $s^0 \in S$  is the initial state;
- $\Sigma$  is a finite set of events, consisting of three mutually disjoint sets of input events  $\Sigma^I$ , output events  $\Sigma^O$ , and internal events  $\Sigma^H$ ;
- $\rightarrow_A \subseteq S \times \Sigma \times S$  is a transition relation. We write  $s \xrightarrow{e} s'$  to denote a transition  $(s, e, s') \in \rightarrow_A$  from state  $s$  to state  $s'$  when event  $e$  occurs.

An input event  $e \in \Sigma^I$  is called *enabled* at a state  $s \in S$  if there is a state  $s' \in S$  such that  $s \xrightarrow{e} s'$ . Otherwise, it is said to be *refused*. We let  $\text{en}^I(s)$  be the set of enabled input events at state  $s$ . An *execution fragment* of  $A$  is a finite alternating sequence of states and events  $s_1, e_2, s_2, \dots, e_k, s_k$  such that  $\forall j: 2 \leq j \leq k, s_1, s_j \in S, e_j \in \Sigma, s_{j-1} \xrightarrow{e_j} s_j$ . In the following we assume a universal set  $\mathcal{U}^{ia}$  of IAs.

The information contained in an IA 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 event  $e \in \Sigma^I$  if the automaton is in a state  $s \in S$  and  $e$  is refused at  $s$ . Also, when the automaton wishes to produce an output event  $e \in \Sigma^O$ , the environment is always ready to accept the output.

Definitions 1 and 2 indicates a similarity in behaviour between IAs and DECs. 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. Put differently, we shall use the terms “event” and “port” interchangeably when relating behaviour of IAs and DECs. Abstracting away data values associated with an event, IAs are suitable for specifying the desirable interaction protocols of DECs with the environment.

The association of IAs with DECs makes it possible to describe the behaviour of components at a high level of abstraction and can thus simplify the analysis of system architectural models, *e.g.* dataflow process networks in this case. It is also very useful for system analysis since component models are often not available when designing system architectures.

As we are only interested in the interaction protocol of DECs, in contrast to definition 2, we will from now on assume that all IAs associated with DECs are deterministic and have no internal transitions. This is justified by the fact that internal transitions do not influence the consistency of a network and also such automata can be transformed into equivalent deterministic automata with no internal transitions (Meduna 2000). In addition, like (Rajamani & Rehof 2002, Yellin & Storm 1997), we exclude automata with mixed states, states where both input and output transitions originate.

Figure 2 shows two examples of IAs. The assumption described by the automaton in figure 2(a) is 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 produce a “payable” message indicating that the customer can pay for the purchase or a “deficit” message indicating otherwise. The automaton in figure 2(b) states that the bank will produce either a “payable” or a “deficit” message but definitely not a “paid” message immediately after receiving a purchase request.

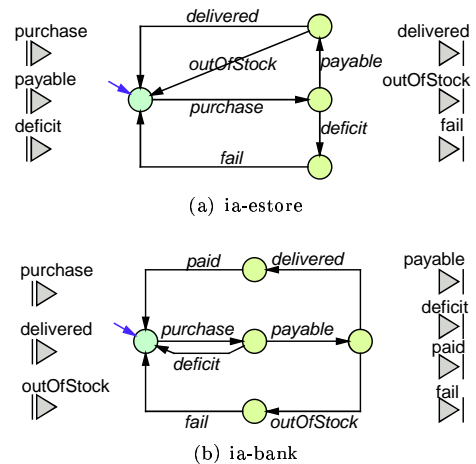


Figure 2: Two interface automata

### 3.3 Consistency of DECs

The association of interface automata with discrete-event components leads to an important issue – the consistency of DECs with respect to IAs. The consistency refers to the fact that a DEC never generates any output which is not allowed by its associated IA if the environment behaves as expected by the IA. If this holds, then in any application context where the IA does not violate a safety property such as the consistency of the composed system, neither does the component. On the other hand, being input-universal, components are able to handle all possible inputs in any state. In other words, components can accept more inputs than IAs. Hence, the consistency cannot be defined by traditional refinement relations, *e.g.* trace containment and simulation, where the implementation can only have less input and output behaviour than the specification.

For this reason, we adopt alternating simulation (de Alfaro & Henzinger 2001) to define the 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 defines the refinement between two IAs, where no data values are involved. We extend this refinement relation to accommodate the implementation (or DECs) with data values.

**Definition 3.** Consider an IA  $A$  and a DEC  $C$  such that  $\Sigma_A^I \subseteq \Sigma_C^I$  and  $\Sigma_A^O \supseteq \Sigma_C^O$ .  $C$  *refines*  $A$  by alternating simulation, written  $C \preceq A$ , if there exists a relation  $\preceq$  such that  $s_C^0 \preceq s_A^0$  and if  $q \preceq s$  for  $q \in S_C, s \in S_A$ , then  $\forall (e, v) \in (\text{en}_A^I(s) \cup \Sigma_A^O) \times \mathcal{U}^{val}$ ,  $\exists q' \in S_C, q \xrightarrow{(e,v)}_C q'$  implies  $\exists s' \in S_A, s \xrightarrow{e} s' \wedge q' \preceq s'$ .

A DEC  $C$  refines an IA  $A$  if and only if  $C$  conforms to the output guarantee of  $A$  with the same helpful environment. In other words, the environment generates an input  $\langle e, v \rangle$  to  $C$  at a state  $q \in S_C$  only when  $A$  at a state  $s \in S_A$  (such that  $q \preceq s$ ) is able to accept the input event  $e$ . Also, conforming to the output guarantee of  $A$  implies that every output event allowed by  $C$  at  $q$  must also be allowed by  $A$  at  $s$ . The definition implies an input and output duality that  $C$  at state  $q$  allows more input events but generates less output events than  $A$  at state  $s$ . It is worth noting that the set of enabled input events of  $C$  at any state always equals  $\Sigma_C^I$ , because  $C$  is input-universal. Hence, an event enabled in  $A$  is always enabled in  $C$ , regardless of the current states of  $A$  and  $C$ .

Definition 3 allows DECs with equal or less output ports to be the implementation of an IA. However, DECs 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 DECs and relax the conditions of definition 3 in defining the consistency of DECs with IAs. Note in the following definition that  $\widehat{C}^\mathcal{O} = C$  if  $\Sigma_C^O \subseteq \mathcal{O}$ .

**Definition 4.** Given a DEC  $C = (s^0, S, \Sigma, \rightarrow)$  and a set  $\mathcal{O}$  of output ports, the *instantiated component* of  $C$  with respect to  $\mathcal{O}$  is defined by  $\widehat{C}^\mathcal{O} = (s^0, S, \Sigma_{\widehat{C}^\mathcal{O}}, \rightarrow_{\widehat{C}^\mathcal{O}})$ , where  $\Sigma_{\widehat{C}^\mathcal{O}}^I = \Sigma^I$ ,  $\Sigma_{\widehat{C}^\mathcal{O}}^O = \Sigma^O \cap \mathcal{O}$ ,  $\rightarrow_{\widehat{C}^\mathcal{O}} = \{(s, \lambda(l), s') \mid (s, l, s') \in \rightarrow\}$ , and  $\lambda(l)$  returns  $\tau$  if  $l \in (\Sigma^O \setminus \mathcal{O}) \times \mathcal{U}^{val}$  or  $l$  otherwise.

**Definition 5.** Consider an IA  $A$  and a DEC  $C$  such that  $\Sigma_A^I \subseteq \Sigma_C^I$ .  $C$  is *consistent* with  $A$  if  $\widehat{C}^{\Sigma_A^O} \preceq A$ .

An algorithm was given in (de Alfaro & Henzinger 2001) for checking alternating simulation between two IAs. However, its first step requires the construction of the Cartesian product of the state spaces of the two automata. Hence, it cannot be applied to checking the consistency of DECs with IAs, because DECs generally have infinite state spaces due to their built-in buffers. Instead, we present a method to cope with this by utilizing the environmental assumptions captured by the specification IAs. The method is simpler than the above algorithm, however it must be ensured that no mixed states are present in the specifications.

### 3.4 Derived interface automata

Before presenting the method, we need to have two auxiliary definitions – mirrors and input-universal versions 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 under alternating simulation defined by (de Alfaro & Henzinger 2001). In addition, we make explicit the environmental assumptions of IAs by building their input-universal versions, where a refused event will lead to an error state.

**Definition 6.** Consider a deterministic IA  $A$  such that  $\Sigma_A^H = \emptyset$ . The *mirror* of  $A$  is an IA  $M = (s_A^0, S_A, \Sigma_M, \rightarrow_A)$ , where  $\Sigma_M^I = \Sigma_A^O$ ,  $\Sigma_M^O = \Sigma_A^I$ , and  $\Sigma_M^H = \emptyset$ .

**Definition 7.** Consider a deterministic IA  $A$  such that  $\Sigma_A^H = \emptyset$ . The *input-universal version* of  $A$  is defined as an IA  $U = (s_A^0, S_A \cup \{\perp\}, \Sigma_A, \rightarrow \rightarrow)$ , where  $\rightarrow \rightarrow = \rightarrow_A \cup \{(\perp, e, \perp) \mid e \in \Sigma_A^I\} \cup \{(s, e, \perp) \mid s \in S_A, e \in \Sigma_A^I, \nexists (s, e, s') \in \rightarrow_A\}$ .

In other words, given a deterministic IA with no internal events, its mirror is constructed by interchanging its input and output events. Also, its input-universal version 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 the refused input events at  $s$ . As input-universal automata are also defined as IAs, to distinguish them from others, we shall call them input-universal automata, while keeping the name “interface automata” for those without  $\perp$  in their state spaces.

As an example, the derived IAs of the “ia-estore” automaton of figure 2(a) are shown in figure 3, where the white triangle “ $\Delta$ ” represents the error state and “\*” matches any of the input events of the automata.

### 3.5 Practical consistency checking of DECs

In this section, the method of checking consistency of DECs with IAs is presented. Firstly, the mirror of a given specification IA is constructed. Next, the product of a given DEC and the mirror is built. Here, an output event of the mirror matches all possible transitions of the DEC accepting an input with an arbitrary

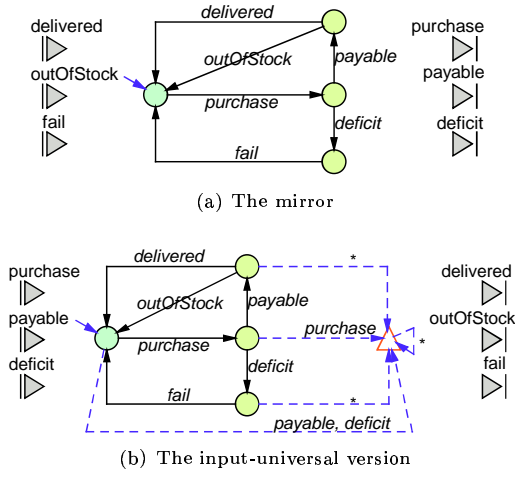


Figure 3: The derived IAs of “ia-estore”

data value. Similarly, an output of the DEC with an associated data value matches an input event of the mirror. Then the consistency of the DEC with the IA is determined by the absence of reachable error states in the product. This approach is justified by theorem 1 (below). Note that in contrast to the algorithm proposed in (de Alfaro & Henzinger 2001), this procedure need only build the reachable states in the product.

**Definition 8.** Consider a DEC  $C$  and an IA  $M$  such that  $\perp \notin S_M$  and  $\Sigma_M^O \subseteq \Sigma_C^I$ . Let  $U$  be the input-universal version of  $M$ . Then the *product* of  $C$  and  $M$  is a DEC  $C_\otimes = (s_\otimes^0, S_\otimes, \emptyset, \rightarrow_\otimes)$ , where:

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

**Theorem 1.** A DEC  $C$  is consistent with an IA  $A$  if  $\Sigma_A^I \subseteq \Sigma_C^I$  and no error state is reachable in the product  $C_\otimes$  of  $C$  and the mirror of  $A$ , i.e.  $\forall \langle q, s \rangle \in S_\otimes, s \neq \perp$ .

*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 an alternating simulation between  $\hat{C}$  and  $A$  by induction. Let  $M$  be the mirror of  $A$  and  $U$  be the input-universal version of  $M$ . First, because  $s_A^0 = s_U^0 \wedge s_C^0 = s_C^0 \wedge (s_C^0, s_U^0) \in \phi$ , we have  $(s_C^0, s_A^0) \in \phi$ . Next, suppose  $\langle q, s \rangle \in \phi$ .

1. If  $q \xrightarrow{\tau}_C q'$  or  $\exists (e, v) \in (\Sigma_C^O \setminus \Sigma_M^I) \times U^{val}, q \xrightarrow{\langle e, v \rangle}_C q'$ , then  $q \xrightarrow{\tau}_{\hat{C}} q'$ . Thus  $\langle q, s \rangle \xrightarrow{\tau}_\otimes \langle q', s \rangle$  and  $\langle q', s \rangle \in \phi$ ;
2. For  $\langle e, v \rangle \in (\text{en}_A^I(s) \cup \Sigma_A^O) \times U^{val}$ , if  $\exists q' \in S_C$  such that  $q \xrightarrow{\langle e, v \rangle}_C q'$ , then  $q' \in S_{\hat{C}} \wedge q \xrightarrow{\langle e, v \rangle}_{\hat{C}} q'$ . For  $e \in \text{en}_A^I(s), \exists s' \in S_A, s \xrightarrow{e}_A s', \text{ i.e. } s \xrightarrow{-e}_U s'$ . For  $e \in \Sigma_A^O, \exists s' \in S_U, s \xrightarrow{-e}_U s'$  since  $U$  is input-universal w.r.t.  $\Sigma_A^O$ . In both cases,  $\exists \langle q, s \rangle \xrightarrow{\tau}_\otimes$

$\langle q', s' \rangle$ . Due to the absence of error states in  $S_\otimes$ ,  $s' \in S_A$  and thus  $\langle q', s' \rangle \in \phi$  holds.

Therefore,  $\phi$  is a relation between  $\hat{C}$  and  $A$  satisfying the conditions of definition 3. Hence,  $C$  is consistent with  $A$ .  $\square$

Now we are able to check the consistency of the “e-store” component of figure 1 with respect to the “ia-estore” automaton of figure 2(a). We calculate the product of the component model and the automaton’s mirror and check for error states in the product. If we find no such error states, then theorem 1 allows us to conclude that the “e-store” component is consistent with the “ia-estore” automaton. At the time of writing, the implementation of this algorithm in the context of Moses is well-advanced.

#### 4 Consistency of component networks

In this section, we extend the work on consistency to networks of components, where the structural information of networks is utilised to facilitate consistency checking. First of all, the language of dataflow process networks (DPNs) used in this paper is defined. A DPN model graphically specifies a DEC by interconnecting a collection of DEC’s via ports. The interpretation of DPNs in terms of DEC’s is also given, which involves not only the synchronization of communication among its component DEC’s but also the interleaving of their internal steps. Hence, the state space of a DPN may grow exponentially with the number of its components.

In order to alleviate this state space explosion problem, the consistency of an open DPN with an IA specifying the interaction protocol of the DPN is not analysed directly. Instead, the IAs with which its component DEC’s prove to be consistent are utilized to determine the consistency of the DPN. This involves the construction of an interface automaton network (IAN), which consists of these IAs and shares the network structure of the DPN, and ensuring the consistency of the IAN with respect to the specification IA. As IAs generally have much smaller state space than DEC’s, this approach is much cheaper. Also, this approach is justified by theorem 2 given in section 4.4.

In addition, with no input or output ports, there is no need to associate an IA with a closed DPN. Its consistency on external behaviour always holds. However, for a closed DPN (or system), system designers are often more concerned about whether its components communicate as designed and whether some invariants can be violated. Often, some safety properties formalizing these requirements have to be provided separately so that confidence about the design can be obtained by model-checking the system against them. Clearly, the state space explosion problem may also occur. The proposed approach can encode some safety properties into IAs associated with system components. By checking the consistency of the network of these IAs, this can help predict the internal interaction behaviour of the system.

##### 4.1 Dataflow process networks

There are many kinds of process networks, such as Kahn’s process networks (Kahn 1974), Karp and Miller Computation Graphs (Karp & Miller 1966), and dataflow process networks (Lee & Parks 1995, Skillcorn 1991). In this paper we consider the form proposed in (Skillcorn 1991).

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 components, but we consider that the buffers are encapsulated in their destination components and the channels represent only the causality of data flow between components. Due to the localization of buffers, the semantic definition of process networks is simplified and thus facilitates modular consistency analysis. Furthermore, it also gives us the flexibility to model a variety of buffers thanks to the diversity of component modelling formalisms.

**Definition 9.** A *network structure* is defined by  $G = (P, \alpha, \Sigma, R)$ , where:

- $P$  is a set of placeholders (or nodes);
- $\alpha$  consists of two functions  $\alpha^I: P \rightarrow \wp(\mathcal{U}^{port})$  and  $\alpha^O: P \rightarrow \wp(\mathcal{U}^{port})$ , mapping from all placeholders to their input and output ports respectively, where  $\wp$  is the powerset operator. We let  $\alpha_P^I = \bigcup_{p \in P} \alpha^I(p)$  be the set of input ports of all placeholders and  $\alpha_P^O = \bigcup_{p \in P} \alpha^O(p)$  be the set of output ports of all placeholders.  $\alpha_P^I$  and  $\alpha_P^O$  can be thought as the set of internal output ports and of internal input ports of the network, respectively.
- $\Sigma \subset \mathcal{U}^{port}$  is a finite set of external ports of the network, consisting of two disjoint sets of input ports  $\Sigma^I$  and output ports  $\Sigma^O$ .
- $R \subseteq (\Sigma^I \times \alpha_P^I) \cup (\alpha_P^O \times \alpha_P^I) \cup (\alpha_P^O \times \Sigma^O)$  is a set of connections, relating internal or external inputs with internal or external outputs, such that the following well-formed requirements hold:
  - no self loops, *i.e.*  $(o, i) \in R \cap (\alpha_P^O \times \alpha_P^I)$  implies  $\rho(o) \neq \rho(i)$ ;
  - no duplicated output to the same placeholder, *i.e.*  $(o, i), (o, i') \in R \wedge i' \neq i$  implies  $\rho(i') \neq \rho(i)$ ;

$$\text{where } \rho(e) = \begin{cases} p \in P & \text{if } e \in \alpha(p) \\ Env & \text{if } e \in \Sigma^I \cup \Sigma^O. \end{cases}$$

The function  $\rho(e)$  returns the placeholder or *Env* associated with the port  $e$ , where *Env* represents the environment of a network with a structure  $G$ . The input and output ports of *Env* corresponds to the output and input ports of the network, respectively. The first well-formed condition claims that there must not be a connection from an output port of a placeholder to any of its own input ports. The second condition requires that any output port of one placeholder (or the environment) should not be connected more than one input port of any other placeholder (or the environment).

**Definition 10.** Given a network structure  $G$  and an instantiation function  $\mathcal{C}: P \rightarrow \mathcal{U}^{dec}$  mapping every placeholder  $p$  to a DEC  $c$  such that  $\alpha^I(p) \subseteq \Sigma_c^I$ , a *dataflow process network (DPN)* is defined as a tuple  $D = (Q, \Sigma, R)$  where  $Q = \{\mathcal{C}(p) \mid p \in P\}$ , and  $\Sigma$  and  $R$  are the same as for  $G$ .

A DPN is called *closed* if  $\Sigma^I = \Sigma^O = \emptyset$ ; otherwise, it is said to be *open*. Graphically, a DPN is depicted as a directed graph. At this level of abstraction, each node represents a placeholder, each triangle associated with a node represents an input or output port of the node, and each edge represents a connection between ports. When the DPN is being constructed at runtime, the component associated with a placeholder

by  $\mathcal{C}$  is instantiated and substitutes the placeholder. Meanwhile, the ports of the placeholder are substituted by the ports of the component with equivalent names. Correspondingly, connections are then established between concrete ports of these components.

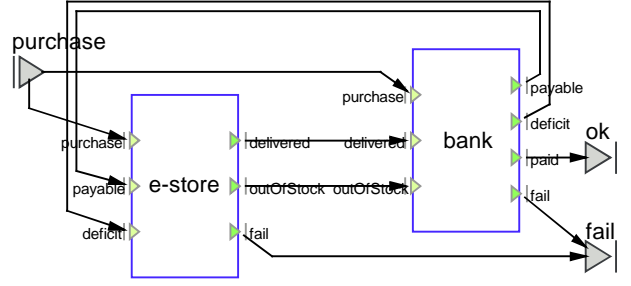


Figure 4: An online purchase DPN

Figure 4 shows the network structure of an online purchase DPN. The instantiation function maps “e-store” to the Petri Net model of figure 1 and “bank” to a bank model (omitted for the sake of brevity). This model accepts purchase requests from customers and reports back whether the purchase succeeds or fails. In order to avoid the complexity of different data values, we assume that the online store sells only one kind of goods. Also, as it is open, this network can be instantiated in a larger DPN including the customer.

DPNs are considered as a special case of DEC. Hence their semantics can be interpreted in terms of DEC, given by the following definition.

**Definition 11.** Consider a DPN  $D = (Q, \Sigma, R)$ . Let  $n = |Q|$ ,  $1 \leq j \leq n$  and  $C_j \in Q$ . Then the DEC represented by  $D$  is defined by  $C = (s^0, S, \Sigma, \rightarrow_C)$ , where

- $S \subseteq S_{C_1} \times \dots \times S_{C_n}$  and  $s^0 = (s_{C_1}^0, \dots, s_{C_n}^0)$ ;
- $\rightarrow_C$  consists of
  - ★ input transitions
$$\{(s, \langle i, v \rangle, s') \mid \langle i, v \rangle \in \Sigma^I \times \mathcal{U}^{val}, \forall j, s'_j = \delta(s_j, i, v)\}$$
  - ★ output transitions
$$\{(s, \langle o, v \rangle, s') \mid \langle o, v \rangle \in \Sigma^O \times \mathcal{U}^{val}, 1 \leq k \leq n, \\ e \in \Sigma_{C_k}^O, (e, o) \in R \wedge s_k \xrightarrow{\langle e, v \rangle}_{C_k} s'_k \\ \wedge \forall j \neq k, s'_j = \delta(s_j, e, v)\},$$
  - ★ internal transitions
$$\{(s, \tau, \langle s_1, \dots, s_j, \dots, s_n \rangle) \mid s_j \xrightarrow{\tau}_{C_j} s'_j\}$$

where  $s = \langle s_1, \dots, s_n \rangle$ ,  $s' = \langle s'_1, \dots, s'_n \rangle$ , and

$$\delta(s_j, i, v) \equiv \begin{cases} q & \text{if } \exists f \in \Sigma_{C_j}^I, q \in S_{C_j}, \\ & (i, f) \in R \wedge s_j \xrightarrow{\langle f, v \rangle}_{C_j} q \\ s_j & \text{otherwise} \end{cases}$$

A state of a DPN is a vector of states of all its components, and its initial state is a vector of their initial states. A DPN is executed by simply executing its components and directing data flow according to the connections  $R$ . An input transition of the network is a transition accepting data at an input port of the network. The transition may involve the synchronization of multiple input transitions of the components,

depending on  $R$ . For example, the input transition at port “purchase” of figure 4 consists of two simultaneous input transitions at the “purchase” ports of the bank and the e-store. In the definition, the function  $\delta(s_j, i, v)$  returns the successor state  $q$  of  $s_j$  if an input port of  $C_j$  is connected with the internal or external input port  $i$ . Otherwise, it returns  $s_j$ .

Similarly, an output transition of the network is a transition resulting in data flow through an output port of the network. The transition may be a single output transition of a component or may involve the synchronization of an output transition of a component and multiple input transitions of other components. For example, the output transition at port “ok” is the output transition of the bank at port “paid”.

Finally, an internal transition of the network is either an internal transition or an output transition of a component. The latter may result in simultaneous data acceptance by other components, but has no external effects such as causing data flow at the network boundary. For example, the output transition of the bank at its port “payable” is internal to the network. Also, it is synchronously executed with the input transition of the e-store at its port “payable”.

Hence, an execution fragment of a DPN can also be understood to be a finite alternating sequence of states and labels as defined in section 3.1 for DEC’s.

From the definition, one can see that, in the execution of a DPN, data can be relayed between components in three ways (Note that we regard here the environment as a special component, the input or output ports of which correspond to output or input ports of the DPN, respectively.) First, when there is only one connection outgoing from an output port of a component, *e.g.* the connection between the “payable” ports of the bank and the e-store, data from the source port is simply directed to the destination port of the connection. Second, when more than one connection originates from one output port, *e.g.* the two connections starting from the port “purchase” of the network, data from the source port is duplicated into multiple destination ports simultaneously. Last, when an input port is connected by multiple output ports, *e.g.* the connections ending at the port “fail” of the network, data from these output ports is merged into one data flow at the input port. In all these situations, the generation of an output from a component occurs synchronously with the acceptance of inputs by the components connected by  $R$ . In other words, the output transition of the source component is executed atomically with the corresponding input transitions of the involved destination components. Therefore, these cases realize two-party or multi-party synchronization between components.

Note that a component could have more input or output ports than the placeholder it substitutes, *e.g.* the Petri Net component in figure 1 vs. the e-store placeholder in figure 4. From definition 11, one can see that disconnecting an input port implies that no data is received via the port, while disconnecting an output port means that any data sent out via the port is lost. In the latter case, one can consider that there is a sink connected to the output port and consumes the data. More precisely, disconnecting an output port of a DEC makes the associated output transition occur independently, while disconnecting an input port of a DEC disables the associated input transition.

## 4.2 Extended process networks and interface automaton networks

In order to make use of the IAs with which its component DEC’s prove to be consistent, we define extended

process networks to include these IAs and also interface automaton networks for them.

**Definition 12.** An *extended process network (XPN)* is defined as a tuple  $X = (G, \mathcal{C}, \mathcal{A})$ , where  $G$  and  $\mathcal{C}$  are the same as in definitions 9 and 10, respectively, and  $\mathcal{A}: P \rightarrow \mathcal{U}^{ia}$  is a function mapping every placeholder  $p$  to an IA  $a$ , such that  $\alpha^I(p) = \Sigma_a^I$  and  $\alpha^O(p) = \Sigma_a^O$ .

**Definition 13.** The *interface automaton network (IAN)*  $N$  derived from  $X$  is defined by  $N = (W, \Sigma, R)$  where  $W = \{\mathcal{A}(p) \mid p \in P\}$ , and  $\Sigma$  and  $R$  are the same as for  $G$  (definition 9).

Consider the online purchase DPN in figure 4. Its XPN and derived IAN share the same structure with its DPN. The function  $\mathcal{A}$  maps the e-store and bank processes to the automata in figure 2(a) and 2(b) respectively. Thus, the set  $W$  consists of these two automata.

Suppose that we have the automaton in figure 5 specifying the external interaction protocol of the example DPN. Instead of proving the consistency of the DPN with the specification IA directly, we can prove the consistency of the DEC’s with their associated IAs and also the consistency of the above-mentioned IAN with the specification IA. The method to ensure the former has been presented in section 3.3, while the method to ensure the latter will be described in the next section.

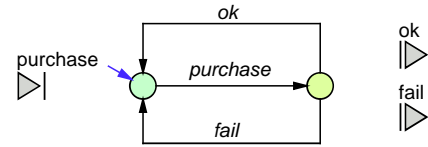


Figure 5: A specification IA “ia-spec”

## 4.3 Consistency of IANs

The consistency of an IAN with a given IA can be informally described as follows (regarding the mirror of the IA as the environment of the IAN): any IA in the IAN or the mirror does not generate an output event which causes an input event violating the assumptions of any other automata (including the mirror). In other words, any legal output will not lead to a refused input event of any other automata (including the mirror). Note that the assumption of the mirror indicates that the IAN never generates more outputs than  $A$  does.

**Definition 14.** Consider an IAN  $N = (W, \Sigma_N, R)$  and an IA  $M$  such that  $\Sigma_M^O \subseteq \Sigma_N^I$ . Let  $n = |W|$ ,  $U_j$  be the input-universal version of  $A_j \in W$  for  $1 \leq j \leq n$ , and  $U_{n+1}$  be the input-universal version of  $M$ . Then the *product* of  $N$  and  $M$  is defined as an IA  $A_\circ = (s_\circ^0, S_\circ, \Sigma_\circ, \rightarrow_\circ)$  such that:

- $s_\circ^0 = \langle s_{U_1}^0, \dots, s_{U_n}^0, s_{U_{n+1}}^0 \rangle$ ;
- $S_\circ \subseteq S_{U_1} \times \dots \times S_{U_n} \times S_{U_{n+1}}$  is the smallest set such that  $s_\circ^0 \in S_\circ$  and  $\forall s \in S_\circ, s \xrightarrow{e}_\circ s'$  implies  $s' \in S_\circ$ ;
- $\Sigma_\circ^I = \Sigma_\circ^O = \emptyset$ , and  $\Sigma_\circ^H = \bigcup_{1 \leq j \leq n+1} \Sigma_{U_j}^O$ ;
- $\rightarrow_\circ = \{ \langle (s_1, \dots, s_{n+1}), e, \langle s'_1, \dots, s'_{n+1} \rangle \rangle \mid 1 \leq j, k \leq n+1, e \in \Sigma_{U_k}^O, s'_k \in S_{U_k}, s_k \xrightarrow{e}_{U_k} s'_k \wedge \forall j \neq k, s'_j = \delta(s_j, e) \}$ , where

$$\delta(s_j, e) \equiv \begin{cases} u & \text{if } \exists i \in \Sigma_{U_j}^I, u \in S_{U_j}, \\ & (e, i) \in R \wedge s_j \xrightarrow{i} u, \\ s_j & \text{otherwise} \end{cases}$$

Note that the product construction uses the input-universal version of each automaton in an IAN or that of the mirror so that any illegal output of an IA can be detected by the reachability of error states. Different from the product construction for DEC and IAs which involves handling data communications of DEC, this procedure for IANs only deals with the interaction protocols of DEC and does not involve data values. Thus, the latter is simpler and cheaper.

**Definition 15.** Consider an IAN  $N$  and an IA  $A$  such that  $\Sigma_A^I \subseteq \Sigma_N^I$ .  $N$  is *consistent* with  $A$  if no error state is reachable in the product of  $N$  and the mirror of  $A$ , i.e.  $\forall \langle s_1, \dots, s_{n+1} \rangle \in S_{\odot}, 1 \leq j \leq n+1, s_j \neq \perp$ .

The consistency of an IAN with an IA is determined by the absence of error states in the product of the IAN and the mirror of the IA. Hence, this guarantees both the compatibility between IAs in the IAN and also the compatibility between the IAN and the mirror, where compatibility between IAs refers to the agreement of the IAs on interaction protocols. Figure 6 shows the product of the example IAN and the mirror of the “ia-spec” automaton of figure 5. As the product state space contains no error state, we can say that the IAN is consistent with the “ia-spec” automaton.

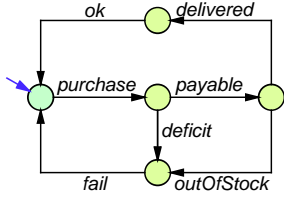


Figure 6: The product of the example IAN and the mirror of “ia-spec”

#### 4.4 Consistency deduction of DPNs

DPNs are a special case of DEC. Hence definition 5 is also applicable to them. The following theorem gives a sufficient condition of determining the consistency of DPNs.

**Theorem 2.** Consider an extended process network  $X = (G, \mathcal{C}, \mathcal{A})$ . Let  $D = (Q, \Sigma, R)$  and  $N = (W, \Sigma, R)$  be its derived DPN and IAN, respectively. Then  $D$  is consistent with an IA  $A$  if

1.  $\Sigma_A^I \subseteq \Sigma^I$ ;
2.  $N$  is consistent with  $A$ ;
3.  $\forall j: 1 \leq j \leq |P|, p_j \in P, A_j = \mathcal{A}(p_j), C_j = \mathcal{C}(p_j)$ , and  $C_j$  is consistent with  $A_j$  (see definition 5).

*Proof.* Let  $n = |P|$ ,  $C_{\otimes}$  be the product of  $D$  and  $M$ , and  $A_{\odot}$  be the product of  $N$  and  $M$ , where  $M$  is the mirror of  $A$ . We shall prove by induction that for all  $\langle q, s \rangle \in S_{\otimes}$ , (a)  $s \neq \perp$ ; (b) Let  $q = \langle q_1, \dots, q_n \rangle$  such that  $\forall j, q_j \in S_{C_j}$ , then  $\exists u_j \in S_{A_j}$  such that  $q_j \preceq u_j$ ,  $u = \langle u_1, \dots, u_n \rangle$  and  $\langle u, s \rangle \in S_{\odot}$ . Clearly, these hold for  $s_{\otimes}^0 = \langle s_D^0, s_A^0 \rangle$  from definitions 3, 5 and 14. Consider  $\langle q, s \rangle \in S_{\otimes}$ . Suppose (a) and (b) hold, then

- i. If  $\exists j, q_j \xrightarrow{\tau} C_j q'_j$ , then  $q \xrightarrow{\tau} D q'$  and  $\langle q', s \rangle \in S_{\otimes}$ , where  $q' = \langle q_1, \dots, q'_j, \dots, q_n \rangle$ . Thus (a) holds. (b) also holds because  $q'_j \preceq u_j$ ;

- ii. If  $\exists \langle e, v \rangle \in \Sigma_M^O \times \mathcal{U}^{val}, s \xrightarrow{e} A s' \wedge q \xrightarrow{\langle e, v \rangle} D q'$ , then for all  $j$ , either  $q'_j = q_j$  or  $\exists \langle e, v \rangle \in \Sigma_{C_j}^I \times \mathcal{U}^{val}, q_j \xrightarrow{\langle e, v \rangle} C_j q'_j$  holds. In the former case, we let  $u'_j = u_j$ . In the latter case,  $e \in \text{en}_{A_j}^I(u_j)$  must hold due to condition 2. Then because  $C_j$  is consistent with  $A_j, \exists u'_j \in S_{A_j}, u_j \xrightarrow{e} A_j u'_j \wedge q'_j \preceq u'_j$ . Let  $u' = \langle u'_1, \dots, u'_n \rangle$ , then  $\exists \langle u, s \rangle \xrightarrow{e} \odot \langle u', s' \rangle$  and  $s' \neq \perp$  (due to definition 14 and condition 2). Clearly, (b) also holds;

- iii. If  $\exists k: 1 \leq k \leq n, \langle e, v \rangle \in \Sigma_{C_k}^O \times \mathcal{U}^{val}, q_k \xrightarrow{\langle e, v \rangle} C_k q'_k$ , then  $\exists \langle q', s' \rangle \in S_{\otimes}$ . Because  $C_j$  is consistent with  $A_j, \exists u'_k \in S_{A_k}, u_k \xrightarrow{e} A_k u'_k \wedge q'_k \preceq u'_k$ . The same as above, we can get  $u'_j$  for all  $j \neq k$  and also  $u'$ . Because  $D$  and  $N$  share the same structure,  $\exists \langle u, s \rangle \xrightarrow{e} \odot \langle u', s' \rangle$ . Hence  $s' \neq \perp$  and (b) holds (due to condition 2).

Therefore,  $\forall \langle q, s \rangle \in S_{\otimes}, s \neq \perp$ . From theorem 1, this theorem holds.  $\square$

With this theorem, we can conclude the example DPN of figure 4 is consistent with the “ia-spec” IA of figure 5, provided that the concrete model of the bank is consistent with the “ia-bank” IA of figure 2(b).

In the context of Moses, we have implemented the check for consistency of an IAN with an IA as specified by condition 2 of theorem 2. This, together with the check based on theorem 1, gives us consistency checking of DPNs.

## 5 Conclusion

In this paper a modular consistency analysis method for dataflow process networks is presented, where interface automata are associated with components to specify the contracts between components and the process networks comprising them. In this way, highly independent development of components and the communication structure among components is supported. Also, a divide-and-conquer approach to checking the consistency of process networks is advocated, where a traditional monolithic checking task is divided into a series of independent tasks at both the architecture level and the component level. As these tasks usually need to handle smaller state spaces than the single monolithic check, the state space explosion problem can be alleviated. Furthermore, on the basis of an optimistic view of the environment, this method can check not only closed process networks but also open ones. This is very helpful for checking the consistency of a subsystem independently from other subsystems and system architectures with which it will be composed.

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 with 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.

In the context of the Moses tool, we have fully implemented the consistency checking of interface automaton networks with the development of a visual notation for interface automata and algorithms and tools for their composition and compatibility checking. The consistency checking of components

with interface automata requires the addition of state space exploration capabilities for heterogeneous components. We are still working on the automation of this checking.

The research presented here is a step towards the automated consistency checking of heterogeneous systems where system components as well as the 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

- Anlauff, M., Kutter, P., Pierantonio, A. & Sünbül, A. (2000), Using domain-specific languages for the realization of component composition, in 'Proceedings of the Fundamental Approaches to Software Engineering (FASE 00)', LNCS 1783, Springer.
- Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R. & Wallnau, K. (2000), Volume II: Technical concepts of component-based software engineering, Technical Report CMU/SEI-2000-TR-008, Software Engineering Institute, Carnegie Mellon University.
- de Alfaro, L. & Henzinger, T. A. (2001), Interface automata, in 'Proceedings of the Foundation of Software Engineering (FSE 01)', Vol. 26 of *Software Engineering Notes*, ACM Press, pp. 109–122.
- Delzanno, G. (2000), Automatic verification of parameterized cache coherence protocols, in 'Proceedings of the Computer Aided Verification (CAV 2000)', LNCS 1855, Springer, pp. 53–68.
- Esser, R. & Janneck, J. W. (2001), Moses - a tool suite for visual modelling of discrete-event systems, in 'Symposium on Visual/Multimedia Approaches to Programming and Software Engineering'.
- Fradet, P., Métayer, D. L. & Périn, M. (1999), Consistency checking for multiple view software architectures, in 'Proceedings of the Foundation of Software Engineering (FSE 99)', LNCS 1687, pp. 410–428.
- Inverardi, P., Wolf, A. L. & Yankelevich, D. (2000), 'Static checking of system behaviors using derived component assumptions', *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9(3), 239–272.
- Janneck, J. W. (2000), Syntax and semantics of graphs - An approach to the specification of visual notations for discrete-event systems, PhD thesis, ETH Zurich.
- Janneck, J. W. & Esser, R. (2002), Higher-order Petri Net modeling—techniques and applications, in 'Proceedings of the workshop on Software Engineering and Formal Methods (ICATPN 02)'.
- Jensen, K. (1997), *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Vol. 1 of *Monographs in Theoretical Computer Science*, Springer-Verlag.
- Jin, Y., Esser, R. & Janneck, J. W. (2002), Describing the syntax and semantics of UML statecharts in a heterogeneous modelling environment, in 'Proceedings of the Diagrammatic Representation and Inference (Diagrams 02)', LNAI 2317, Springer.
- Kahn, G. (1974), The semantics of a simple language for parallel programming, in 'Proceedings of the IFIP Congress 74', North-Holland Publishing Co., pp. 471–475.
- Karp, R. & Miller, R. (1966), 'Properties of a model for parallel computations: determinacy, termination, queuing', *SIAM J. Appl. Math.* 14, 1390–1411.
- Lee, E. A. & Parks, T. M. (1995), 'Dataflow process networks', *Proceedings of the IEEE* 83(5), 773–801.
- Mazurkiewicz, A. (1989), Basic notions of trace theory, in 'Proceedings of the School/Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency', LNCS 354, Springer, pp. 285–363.
- Meduna, A. (2000), *Automata and Languages: Theory and Applications*, Springer.
- Milner, R. (1989), *Communication and Concurrency*, Intl. Series in Computer Science, Prentice Hall.
- Nuseibeh, B., Easterbrook, S. & Russo, A. (2001), 'Making inconsistency respectable in software development', *Journal of Systems and Software* 58(2), 171–180.
- Rajamani, S. K. & Rehof, J. (2002), Conformance checking for models of asynchronous message passing software, in 'Proceedings of the Computer-Aided Verification (CAV 02)', LNCS, Springer.
- Reisig, W. & Rozenberg, G., eds (1998), *Lectures on Petri Nets I: Basic Models*, Advances in Petri Nets, LNCS 1491, Springer-Verlag.
- Skillcorn, D. B. (1991), Stream languages and data-flow, in 'Advanced Topics in Data-Flow Computing', Prentice Hall.
- Sunetnanta, T. & Finkelsteing, A. (2001), Automated consistency checking for multiperspective software specifications, in 'Proceedings of the workshop on Advanced Separation of Concerns (ICSE 01)'.
- Uchitel, S. & Yankelevich, D. (2000), Enhancing architectural mismatch detection with assumptions, in 'Proceedings of the Engineering of Computer Based Systems (ECBS 00)'.
- van der Aalst, W., van Hee, K. & van der Toorn, R. (2002), 'Component-based software architectures: A framework based on inheritance of behavior', *Science of Computer Programming* 42(2-3), 129–171.
- van Glabbeek, R. J. & Weijland, W. P. (1996), 'Branching time and abstraction in bisimulation semantics', *Journal of the ACM* 43(3), 555–600.
- Winskel, G. (1987), Event structures, in 'Petri Nets: applications and relationships to other models of concurrency', LNCS 255, pp. 325–392.
- Yellin, D. M. & Storm, R. E. (1997), 'Protocol specifications and component adaptors', *ACM Transactions on Programming Languages and Systems* 19(2), 292–333.