

# Consistency and Interoperability Checking for Component Interaction Rules

Yan Jin and Jun Han

*Faculty of ICT, Swinburne University of Technology, Hawthorn, VIC 3122, Australia*  
{yjin, jhan}@ict.swin.edu.au

## Abstract

*In component-based software development, it is important to ensure interoperability between components based on their unambiguous semantic descriptions, in order to obtain a viable system. A body of recent work has explored the use of formal languages in specifying component interaction protocols for interoperability checking, but lacks the practicality required by software practitioners for daily use. Faced with this, we have developed a lightweight specification approach to component interaction rules, which has the necessary expressiveness and employs event patterns in rule specification for easy adoption by practitioners. In this paper, we present a FSA-based semantic model for such rules and novel studies of rule consistency and system-wide protocol interoperability for components annotated with interaction rules. We also develop incremental approaches and tools to check these properties, which provide an effective means to discover errors in the design of component interaction rules and component-based system architectures.*

## 1 Introduction

A key feature of component-based software engineering is to allow the construction of an application using independently developed software components, leading to reduced development costs and improved software quality. In this process, it is essential to ensure that individual components can in fact interoperate properly and achieve the desired functionality of the composite system. This requires the understanding and capture of the semantic or behavioural characteristics of a component, including its operation semantics, qualities, and interaction protocols, which are beyond the commercial Interface Definition Languages (IDLs) for components [15].

In particular, the interaction protocol of a component defines the rules governing the interaction with it, *e.g.* the valid sequences of message exchanges or service/operation invocations. Adherence to the protocols by both its implementation and the other system components communicating

with it is essential to enforce its proper use and functioning in a system context as well as protocol interoperability with other components. This is in turn fundamental to achieve higher-level semantic and qualitative interoperability [17] in the system.

The research community has widely recognised the need for precise specification of interaction protocols to enable automated interoperability reasoning, and proposed various formal language based approaches, *e.g.* process algebras, Petri nets, finite state machines, regular expressions, description logic, etc. Although having many advantages, they have not gained wide acceptance in the industry, if at all.

We believe this is mainly due to the fact that software practitioners are not familiar with the formal specification languages, processes, and strategies underlying these approaches. Understanding and developing such formal specifications often requires considerable effort. This is especially true when the component exhibits complex interaction logic. Furthermore, there is no easy way for the practitioners to modify such formal specifications as they could add, delete, or change narrative clauses in informal documentation. It is thus not simple for component developers to control how much information to publish for a given component. The information has to be sufficient to facilitate proper use of components by third parties, while being protective to proprietary implementation techniques. In addition, it can be hard for the practitioners to ensure they have properly applied these formal language based approaches, *e.g.* whether interaction protocols are correctly expressed as intended and whether all important aspects are specified. This requires a deep understanding of the specification language' underlying formality and modelling concepts, *e.g.* abstraction and composition mechanisms. As a consequence, the cost and risk are often considered too high to adopt such a formal language based approach, given the tight project schedule.

In earlier work [19, 20], we have presented a lightweight specification approach and run-time validation tool for component interaction protocols, which combines the practitioner-familiar clause-based description style with formal rigour. The approach is aimed for use by software

practitioners and does not require that the users have profound knowledge of formal methods. Unlike existing formal approaches that encode all the rules governing interaction with a component into a process model, this approach specifies these rules individually and declaratively as they were in informal documentation. More specifically, these rules are specified as *interaction constraints* (ICs) using event patterns proposed by Dwyer *et al.*[11, 12] (with our own extensions). They may concern with occurrences or sequencing of operation invocations from/to the component. Due to the simplicity and intuition of the event patterns, the specified ICs can be easily read as narrative clauses. Accordingly, a simple correspondence with informal documentation is easily established.

In this paper, we present a systematic approach to defining the semantics of interaction constraints in terms of finite state automata (FSAs) for formal analysis. On this basis, we give novel definitions for both interaction rule consistency and system-wide protocol interoperability for components with specified ICs. The consistency definition is based on the intention of component developers in designing component interface signature, and makes explicit the interaction between the interaction rule design and the component interface design. The interoperability definition takes into account the intention of system designers in organising the communication structure between components, and enables independent, optimistic design and analysis of open systems. Such explicit definitions are currently missing in the literature. To provide an effective means to discover errors, we propose incremental approaches to checking the consistency and interoperability. These checking approaches have been implemented in a prototype software tool.

The rest of this paper is structured as follows. After an overview of our specification approach, we present a semantic model for interaction rules in Section 3. We then define the concepts of rule consistency and protocol interoperability for components, and propose incremental checking approaches in Section 4 and 5, respectively. Following a brief description of our tool implementation in Section 6, we present related work and conclusions in Section 7 and 8.

## 2 Pattern-based specification of component interaction rules

**A motivating example.** Let us consider an auctioneer component in a distributed auction system, originally taken from [6]. It can hold auctions among registered bidders, by providing three operations for registration, unregistration and bidding, and using three operations of bidders for bidding announcement, bidding settlement, and auction closing. Figure 1 shows the IDL definitions for auctioneers and bidders, where *auctNo* uniquely identifies an auction.

Clearly, semantic information about the auctioneer to en-

```

1 interface IAuctioneer {
2   void register(in IBidder b);
3   void unregister(in IBidder b);
4   boolean bid(in long auctNo, in float price);
5 }
6 interface IBidder {
7   oneway void openForBid(in long auctNo,
8     in string itemDesc, in float price);
9   void youGotIt(in long auctNo, in float price);
10  oneway void auctionClosed(in long auctNo);
11 }

```

Figure 1. IDL definitions

able system behavioural analysis is missing in these IDL definitions. The most fundamental are descriptions about the rules governing its interaction with bidders. The rules should describe how its services can be utilised and what are the obligations of both the auctioneer and the bidders when the auction services are being used. For example, how can bidders participate in auctions? How does the auctioneer conduct auctions? What will eventually happen to an item for which there exists a bid? Will abnormal behaviour be possible, e.g. selling an item to a bidder who did not bid? Unambiguous answers to such questions are critical for the bidder components to interact properly with the auctioneer component, resulting in a viable auction system.

**Interaction rule specification.** To address these issues, we have proposed in [19, 20] a pattern-based specification approach to such rules. A single rule on the occurrences or sequencing of message exchanges and/or operation invocations is defined as an *interaction constraint* (IC) using easily-understood event patterns from the Specification Pattern System (SPS) [10, 11, 12] with our own extensions. Due to its limited scope, specifying a single IC is much easier than specifying the overall interaction protocol.

Figure 2 presents some example rule specifications for the auctioneer component. Identity-aware rules on neighbouring components are declared at the peer level, while the others are at the component level. Each of these rules applies a pattern (with a scope) to a number of operations.

```

1 component Auctioneer {
2   provides IAuctioneer;
3   requires IBidder;
4
5   interaction-contract {
6     peer-level:
7     bid exists only after register until unregister;
8     bid precedes youGotIt globally
9     where bid.auctNo = youGotIt.auctNo;
10    ...
11   component-level:
12   bid leads to youGotIt globally
13   where bid.auctNo = youGotIt.auctNo;
14   ...
15 }
16 }

```

Figure 2. Interaction rules for Auctioneer

The specific operation invocations of interest and the relationship between their parameter/return values are specified by the associated **where** clause. In particular, the first peer-level IC applies to each individual bidder and states the relative sequencing among invocations to *register*, *bid* and *unregister* from/to a bidder, regardless of the parameter values. Essentially, it means that only registered bidders can make a bid. The second peer-level IC requires that if the *youGotIt* operation of a bidder is invoked regarding an auction *auctNo*, it must be preceded by a bid of that bidder for *auctNo*. That is, an item being auctioned can only be sold to a bidder who has bid for it. Furthermore, the component-level IC states the causality between *bid* and *youGotIt*, i.e. an item being auctioned will be sold if a bid is made. We note it enforces no identity correspondence between the bidders who bid and the bidder who finally gets the item.

As seen above, the specification of component interaction protocols takes a divide-and-conquer approach and is based on easily-understood event patterns. These patterns can easily be translated into natural languages for better understandability (Cf. [23]). Furthermore, it is easy to add more ICs to make the specified component interaction behaviour more deterministic. For instance, the auctioneer is required not to sell an item to two or more bidders at an auction. One can add a component-level IC, stating that *youGotIt* invocations can occur at most once with respect to each given auction *auctNo*. All these features ease the protocol specification for the component developers and the understanding process for the component user. These also enable incremental evolution of component interaction protocols.

### 3 Semantics of interaction rules

In order to enable interoperability analysis, it is essential to define a formal semantics for interaction constraints. In this section, we present a FSA-based semantic model for ICs in two steps: (1) formalise SPS event patterns for IC specification as FSA templates with transition labels being formal event set parameters; and (2) for a given IC, instantiate a number of FSAs from the used pattern with different actual event sets for the labels. Such a FSA corresponds to each possible value combination of the referenced parameters and neighbouring component (if applicable).

#### 3.1 Interaction constraint automata

To facilitate later presentations, we define *interaction constraint automata* (ICA) as a special case of FSAs, assuming a finite set  $\Sigma$  of interface events for the component of interest, with  $\Sigma = \Sigma^I \cup \Sigma^O$ ,  $\Sigma^I \cap \Sigma^O = \emptyset$ ,  $\Sigma^I$  containing input events, and  $\Sigma^O$  output events. Each such event identifies a producer, a consumer component, and a message.

**Definition 1.** Given a component with a finite set of interface events  $\Sigma$ , an *interaction constraint automata* is a FSA  $A = (q^0, Q, F, \Sigma, \delta)$ , where  $Q$  is a finite set of states,  $q^0 \in Q$  is the initial state,  $F \subseteq Q$  is a set of final states, and  $\delta: Q \times \Sigma \rightarrow Q$  is a partial transition function.

Given a FSA  $A$ , an event  $e \in \Sigma$  is said to be *rejected* at a state  $q \in Q$  if  $\delta(q, e)$  is undefined. Otherwise,  $e$  is *accepted* at  $q$ . A finite event sequence  $\sigma \in \Sigma^*$  is called a *trace of A from q* if, let  $\sigma = e_1 e_2 \dots e_n$ , then  $\exists q_1, \dots, q_n \in Q$  such that  $\delta(q, e_1) = q_1, \delta(q_1, e_2) = q_2, \dots, \delta(q_{n-1}, e_n) = q_n$ . There,  $q_n$  is called *reachable from q via  $\sigma$* . It is said to be *accepted by A* if  $q = q^0 \wedge q_n \in F$ . The language  $\mathcal{L}(A)$  of  $A$  is a set consisting of all accepted traces of  $A$ .  $A$  is *well-formed* if (1) every state in  $Q$  is reachable from  $q^0$ ; (2) it contains no deadlock or livelock state. A state  $q \in Q$  is *deadlocked* if  $q \notin F$  and  $\forall e \in \Sigma, \delta(q, e)$  is undefined.  $q$  is *livelocked* if  $q \notin F$ ,  $q$  is not deadlocked, and  $\nexists q' \in F$  s.t.  $q'$  is reachable from  $q$ . We ensure all ICAs are well-formed.

#### 3.2 FSA templates for patterns

As a first step towards formalising ICs, we semantically define SPS event patterns using FSA templates. The templates are derived, using standard transformation methods, from the regular expressions defined in [10]. They are illustrated by three patterns used in Figure 2, assuming  $\Sigma$  is the set of all interface events of the component, and  $E_1, E_2, E_3 \subset \Sigma$  are mutually disjoint sets. Such a FSA transits between states when any event in a labelling set of a transition occurs. Its transitions are labelled with formal event sets to cater for user-specified operands.

Pattern “ $E_1$  precedes  $E_2$  globally” is represented by Figure 3(a), where  $O = \Sigma \setminus (E_1 \cup E_2)$  denotes all the other events. In essence, it requires that any event occurrence in  $E_2$  be preceded by at least one event occurrence in  $E_1$  in the whole interaction history with the component. Pattern “ $E_1$  leads to  $E_2$  globally” is depicted by Figure 3(b) with  $O = \Sigma \setminus (E_1 \cup E_2)$ . It states that after any event occurrence in  $E_1$ , there will eventually be an event occurrence in  $E_2$  (before the component terminates). Pattern “ $E_2$  exists only after  $E_1$  until  $E_3$ ”, represented by Figure 3(c) with  $O = \Sigma \setminus (E_1 \cup E_2 \cup E_3)$ , states any event occurrence in  $E_2$  is possible only when an event in  $E_1$  has occurred but no event in  $E_3$  has occurred afterwards.

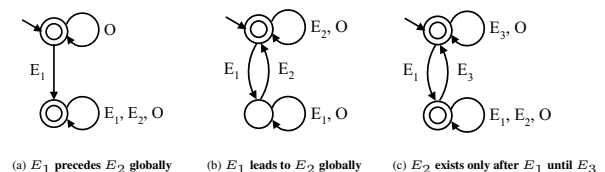


Figure 3. FSA templates for example patterns

### 3.3 Formalising interaction constraints

In this section, we show how to derive the actual event sets to obtain the ICAs for an IC. Such an ICA is instantiated from the corresponding FSA template for each value combination of operation parameters and neighbouring components (peer-level ICs only). In a peel-level ICA, we let the actual event sets for all pattern operands include only events regarding a same neighbour, whereas in a component-level ICA, we do not enforce such neighbour identity correspondence. In addition, we include in such a set only certain types of events that are significant to define the boundary with other operands in the interaction timeline. The event types can be either call, return, or both. For instance, for “ $op_1$  precedes  $op_2$ ”, return events of operation  $op_1$  and call events of  $op_2$  will be used, whereas for “ $op_2$  exists only after  $op_1$  until  $op_3$ ”, both call and return events of  $op_2$ , return events of  $op_1$ , and call events of  $op_3$  will be used.

As an example, suppose an auctioneer  $a$  will only conduct two auctions 101 and 102 between bidders  $b_1$ ,  $b_2$  and  $b_3$ . Then, if abstracting away the operation parameters and results, e.g. *price*, which are not referenced in any IC specifications, the three rules directly governing the interaction with  $b_1$  can be written as follows:

```

1 { (b1,a,bid_call,101), (a,b1,bid_retn,101), (b1,a,bid_call,102), (a,b1,bid_retn,102) } exists only after { (b1,a,register_retn,b1) } until { (b1,a,unregister_call,b1) };
2 { (a,b1,bid_retn,101) } precedes { (a,b1,youGotIt_call,101) } globally;
3 { (a,b1,bid_retn,102) } precedes { (a,b1,youGotIt_call,102) } globally;

```

An event is written as a tuple of producer, consumer, event type, parameters, e.g.  $(b_1, a, bid\_call, 101)$ . The peer-level rules applicable to  $b_2$  and  $b_3$  can be obtained similarly. Furthermore, the rules applicable to the interaction with the whole group of bidders can be written as follows (we omit the rules for auction 102 to avoid unnecessary repetition):

```

1 { (b1,a,bid_call,101), (a,b1,bid_retn,101), (b2,a,bid_call,101), ..., (a,b3,bid_retn,101) } exists only after { (a,b1,openForBid,101), ..., (a,b3,openForBid,101) } until { (a,b1,auctionClosed,101), ..., (a,b3,auctionClosed,101) };
2 { (a,b1,bid_retn,101), (a,b2,bid_retn,101), (a,b3,bid_retn,101) } leads to { (a,b1,youGotIt_call,101), (a,b2,youGotIt_call,101), (a,b3,youGotIt_call,101) } globally;
3 ...

```

Based on the rules and actual event sets given above, the method presented in Section 3.2 can be used to construct the corresponding ICAs. All ICAs for  $a$  can thus be obtained.

### 4 Consistency checking for interaction rules

We have so far defined the semantics for single interaction constraints in terms of ICAs. In this section, we examine how to obtain the overall interaction protocol from ICAs, and define the consistency of ICAs in order to provide a guide for IC specification. We further propose an incremental approach to checking such consistency.

**Interaction protocols.** To define protocols at a sufficient level of abstraction, we assume  $\mathcal{P}^0 \subseteq \Sigma^*$  denotes the set of all acceptable event sequences by the component when no IC is considered. This set can include occurrence or sequencing constraints attributing to the physical world or underlying platform, e.g. an operation reply event always occurs after the operation call event that causes it. A treatment of  $\mathcal{P}^0$  is left here as a semantic variation point.

**Definition 2.** Given a component  $c$ , let  $\mathbb{A}$  be a set of well-formed ICAs derived from  $c$ 's IC specifications, then the *interaction protocol*  $\mathcal{P}$  of  $c$  wrt  $\mathbb{A}$  is defined by the language intersection of these ICAs, i.e.  $\mathcal{P} = \mathcal{P}^0 \cap \bigcap_{A \in \mathbb{A}} \mathcal{L}(A)$ . The interaction protocol of  $c$  wrt the set of all derived ICAs is called the *overall interaction protocol* of  $c$ .

The interaction protocol of a component contains the set of all acceptable traces. Each trace has to be accepted by every ICA in  $\mathbb{A}$ . Given a component  $c$  and a set  $\mathbb{A}$  of its derived ICAs, we call a well-formed FSA  $A$  an *interaction protocol automaton* (IPA) of  $c$  wrt  $\mathbb{A}$  if  $\mathcal{L}(A)$  is equal to the interaction protocol of  $c$  wrt  $\mathbb{A}$ . We note all derived ICAs and IPAs have the same set of interface events.

**Interaction rule consistency.** When a component is published, it is generally expected that every provided operation in its interface can be used in certain ways (otherwise there is no point to include it in the interface). We take this into account in defining the consistency of interaction rules in Definition 3 below. More specifically, we require, for each provided operation, there exists at least one event sequence in the interaction protocol where a message associated with the operation appears. This can help identify the situation where calls to an operation are never possible.

**Definition 3.** Let  $c$ ,  $\mathbb{A}$  and  $\mathcal{P}$  be as in Definition 2, and  $\Theta$  be a set of operations provided by  $c$ . Then ICAs in  $\mathbb{A}$  are *consistent* wrt  $\Theta$  if, for each  $op \in \Theta$ , there exists an event sequence  $\sigma \in \mathcal{P}$  such that a call event of  $op$  appear in  $\sigma$ .

This definition enables us to follow a three-step process to check the consistency of the overall protocol of a given component: (1) build all the ICAs from the component IC specifications; (2) use standard algorithms for computing FSA intersection to obtain an IPA whose language represents the interaction protocol; and (3) check in the resultant IPA for satisfaction of the stated consistency condition.

It is worth noting that the rule consistency enforces stronger requirements on accepted traces than interoperability defined in Section 5. This is because the interaction protocol is expected to be comprehensive in interaction scenarios in order to enable the use of the component in a variety of systems, whereas interoperability only concerns with a specific system and some operations may not be used at all.

**Incremental consistency checking.** We observe the interaction protocol is the language intersection of all ICAs. This means disproving rule consistency may not need to compute the overall protocol. The existence of a certain event sequence has to hold on the protocol for any subset of ICAs. This gives us a way to incrementally debug for consistency. In many cases, discovering errors is more useful than proving their absence.

As an example, suppose we add a peer-level IC to the auctioneer: *bid precedes register globally*. To debug for *a*, we first assume a single bidder, say *b*<sub>1</sub>, and examine the consistency between the ICAs for the following rules:

```

1 { (b1,a,bid_call,101), (a,b1,bid_retn,101), (b1,a,bid_call,102), (a,b1,bid_retn,102) } exists only after { (b1,a,register_retn,b1) } until { (b1,a,unregister_call,b1) };
2 { (a,b1,bid_retn,101) } precedes { (b1,a,register_call,b1) } globally;
3 { (a,b1,bid_retn,102) } precedes { (b1,a,register_call,b1) } globally;

```

It is easy to prove that the resulting protocol is not consistent according to Definition 3, since no calls to *bid* or *register* are possible. This in turn disproves the consistency of the overall interaction protocol of *a*, no matter how many bidders are in the system and how many other ICs are specified for the auctioneer. Clearly, checking only the corresponding three ICAs requires much less computation to reveal the inconsistency.

## 5 Interoperability checking for components

Publishing interaction constraints with component interfaces provides rich and unambiguous descriptions about both each component's behavioural assumptions on other components in a system and their behavioural commitments in response to service requests. Such descriptions allows for rigorous reasoning of behavioural or protocol interoperability between the components towards designing a correct functioning system. In this section, we define component protocol interoperability in the system context and propose an incremental approach to checking it. We assume components are compatible in signature, *e.g.* data types.

### 5.1 Composition of interaction protocol automata

To define protocol interoperability, we need a composition mechanism to relate the behaviour of components to the system behaviour. For this purpose, we adopt the asynchronous model of interface automata [8]. Compared with rendezvous synchronisation, this model can reflect more closely the nature of communication in distributed systems, where a component has no control over when an input from the environment comes in. Thus a component cannot block inputs from happening before it is ready to process them. The adoption of this composition mechanism enables optimistic reasoning of partially built (or open) systems and

differentiates our approach from many existing component specification approaches such as [1, 4, 6, 21, 22, 24].

Given a system with a set  $\mathbb{A}$  of IPAs, one for each component, due to the way an interface event is built (Cf. Section 3.1), the IPAs' input (or output) event sets are mutually disjoint. Assuming an unique trap state  $\perp$  for output-input mismatch, the IPA composition is defined as follows:

**Definition 4.** The *composition* of component IPAs in a set  $\mathbb{A}$  is defined by a FSA  $A = (q^0, Q, F, \Sigma, \delta)$  such that  $q^0 = \prod_{A \in \mathbb{A}} q_A^0$ ,  $Q \subseteq \prod_{A \in \mathbb{A}} Q_A \cup \{\perp\}$ ,  $F \subseteq \prod_{A \in \mathbb{A}} F_A$ ,  $\Sigma \subseteq \bigcup_{A \in \mathbb{A}} \Sigma_A$ ,  $n = |\mathbb{A}|$ ,  $1 \leq i, j \leq n$ ,  $i \neq j$ , and

$$\delta(\mathbf{q}, e) = \begin{cases} \text{undefined} & \text{if } \mathbf{q} = \perp \\ \perp & \text{if } \mathbf{q} = (q_1, \dots, q_n) \wedge e \in \Sigma_i^O \cap \Sigma_j^I \\ & \wedge \delta(q_i, e) = q'_i \wedge \delta(q_j, e) \text{ is undefined} \\ (q_1, \dots, q'_i, \dots, q'_j, \dots, q_n) & \text{if } \mathbf{q} = (q_1, \dots, q_n) \wedge i < j \wedge e \in \Sigma_i \cap \Sigma_j \\ & \wedge \delta(q_i, e) = q'_i \wedge \delta(q_j, e) = q'_j \\ (q_1, \dots, q'_i, \dots, q_n) & \text{if } \mathbf{q} = (q_1, \dots, q_n) \wedge \delta(q_i, e) = q'_i \\ & \wedge e \notin \bigcup_{1 \leq k \leq n \wedge i \neq k} \Sigma_k \end{cases}$$

This definition distinguishes between input and output events, and binds an output event produced by a component IPA to the same input event of another component to reflect the asynchronous nature of communications between components. That is, to consume a shared input event, a component has to wait for the other component to produce it as output. In contrast, a component is always allowed to produce an output, no matter whether the other component agrees or not. However, if the other component is not ready to receive it as input, an interoperability problem will occur. This is captured by moving the composed system into the trap state  $\perp$ . Note that  $\perp$  is not a final state and no transition can move the system out of  $\perp$ . Note also that the composed system being in a final state implies all component IPAs being in a final state.

Below, we define three important concepts for the composition: neighbours, communicative ability and compatibility.

**Definition 5.** Let  $\mathbb{A}$  and  $A$  be as in Definition 4. Then two IPAs  $A_1, A_2 \in \mathbb{A}$  are called *neighbours* if  $\Sigma_1 \cap \Sigma_2 \neq \emptyset$ . Two neighbours  $A_1$  and  $A_2$  are said to *be able to communicate in A* if there exists an accepted trace  $e_1 e_2 \dots e_n$  of  $A$  such that  $\Sigma_1 \cap \Sigma_2 \cap \{e_1, \dots, e_n\} \neq \emptyset$ .

Two IPAs (or the components represented by them) are neighbours if they share some interface events. They are able to communicate in the composite system if some shared event appears in an accepted system trace.

**Definition 6.** Let  $\mathbb{A}$  and  $A$  be as in Definition 4 and  $\mathbb{B} \subseteq \mathbb{A}$  be a subset of IPAs. Then the IPAs in  $\mathbb{B}$  are called *compatible* in the system if no error states (such as trap, deadlock, or livelock states) are reachable in  $A$  via a sequence of events from  $\bigcup_{B \in \mathbb{B}} \Sigma_B^O$ .

A subset of IPAs (or the corresponding components) are compatible in the composite system if their composition  $A$  does not contain an accepted trace of output events of these components, which leads the system to an error state. In other words,  $A$  cannot go to such a state without taking an input from components outside this subset. This idea of compatibility is borrowed from [8], reflecting an optimistic view of the environment (composed of the other components in the system). That is, the components can still work together as long as the environment can, by selectively providing inputs, prevent their composed subsystem from entering an error state. This results in optimistic compatibility of components better-suited to analysing open systems.

When it comes to a complete (or closed) system, we know  $\bigcup_{B \in \mathbb{B}} \Sigma_B^O$  equals to the set of events of the whole system. Hence the compatibility of all the IPAs represents a requirement that the state set of the composition contain no error state. If restricting such a set to reachable states, this implies the well-formedness of the composed FSA.

## 5.2 System-wide protocol interoperability

By system-wide interoperability, we are concerned with the adherence of any accepted trace of the system to the interaction rules of all the components, resulting in a compatible system. We also requires the existence of scenarios where any given component is able to engage in communications with each of its neighbours. More precisely, for each pair of neighbouring components, there must exist an accepted trace of the system, where at least one shared interface event between them appears, *i.e.* the components interact and make joined transitions. This allows us to identify situations where compatibility or deadlock freedom fails to identify, *e.g.* dead links (or channels) between components.

**Definition 7.** Consider a system composed of components from a set  $C$ . Let  $\mathbb{A}$  be a set of IPAs (one for each component), and  $A$  be the composition of IPAs in  $\mathbb{A}$ . Then the IPAs in  $\mathbb{A}$  are called *interoperable in the system* if

- Each pair of neighbours can communicate in  $A$ ;
- All IPAs in  $\mathbb{A}$  are compatible in the system.

The components in  $C$  are *interoperable in the system* if their overall IPAs are interoperable in the system.

This definition is applicable to both closed and open systems. In a closed system, the second clause implies that the system never enters the trap state  $\perp$ , since every event is an output of some component. In an open system, the existence of  $\perp$  is tolerated since there are still chances where

a larger system can prevent  $\perp$  from being entered by selectively providing inputs.

**Incremental interoperability checking.** As for consistency checking, very often debugging is more valuable than proving interoperability in a component-based system design, especially in the early design stages. A useful and practical tool for cheaper (and possibly earlier) error detection can be provided by an incremental approach to interoperability checking. The following theorem gives the basis for our incremental approach.

**Theorem 1.** Consider a system composed of components from a set  $C$ . If all the components in  $C$  are interoperable in the system, then all the components in any subset  $C' \subseteq C$  are also interoperable in the formed subsystem.

*Proof.* Suppose the components in some set  $C' \subseteq C$  are not interoperable in the formed subsystem. Then in the composed FSA  $A'$  of the overall protocols of components in  $C'$ , there exists a trace  $e_1, \dots, e_n$  of output events from  $\bigcup_{c \in C'} \Sigma_c^O$  leading to an error state. Because the other components in  $C \setminus C'$  cannot block these events from happening in the system  $C$ ,  $e_1, \dots, e_n$  is also a trace of that system. Clearly, this would lead the system to either a trap, deadlock, or livelock, because a component or subsystem being at a non-final state implies the system being at a non-final state. Therefore, the theorem holds.  $\square$

This theorem gives us a means to detect interoperability problems in a system design, starting from a minimum subsystem of components. It allows one to gradually include more components for debugging and conduct more focussed investigations when problems are identified.

As an example, consider an auction system with an auctioneer  $a$  and a number of bidders, among which  $b_1$  has one peer-level interaction rule:

```
{ (a,b1,auctionClosed,101), (a,b1,auctionClosed,102) }
exists only after { (b1,a,register_retn,b1) } until
{ (b1,a,unregister_call,b1) }.
```

Intuitively, one may suspect the auction system have an interoperability problem, since the interaction rules of  $a$  as specified in Figure 2 do not restrict the occurrence of *auctionClosed* calls in any way and  $a$  may initiate such a call to  $b_1$  when  $b_1$  is at the initial state and unregistered. However, to formally prove this conjecture based on Definition 7 has a high computation cost. The cost can increase dramatically with the number of bidders. In contrast, using Theorem 1, one can check only the interoperability between  $a$  and  $b_1$  to identify the problem. This results in a much cheaper and faster computation, insensitive to the number of other components in the system. We note, although this checking method is effective in suspected error detection, the expensive construction of the full system state space is still needed in cases where the interoperability holds.

## 6 Tool implementation

Based on the work presented above, we have developed a prototype tool for consistency and interoperability checking of component interaction rules. Its main modules are shown in Figure 4. The component ICAs for checking are constructed by an *ICA generator* (ICAG). ICAG utilises two modules for the construction: (1) a *pattern library* (PL), which defines the FSA templates for SPS patterns and scopes; (2) an *interface event set builder* (IESB), which relies on a user-supplied finite set of values for each parameter referenced by some IC specification, and interactively assists the user in building the set of interface events for a component. IESB also automatically removes parameters not referenced in any IC from the event set to reduce the checking complexity. For each component and each of its IC specifications, ICAG constructs a number of ICAs, each corresponding to a value combination of referenced parameters and neighbouring component (peer-level ICs only). It stores the resultant ICAs in a database for checking.

The *rule consistency checker* (RCC) implements standard algorithms for FSA intersection. Based on the generated ICAs, it can compute interaction protocol automata for the given component and check if the consistency condition is satisfied. It also allows the user to check a subset of ICAs and thus provides a useful debugging tool for suspected inconsistency between the specified rules. RCC can be used by both component developers and system designers.

The *protocol interoperability checker* (PIC) implements algorithms for the IPA composition mechanism, communicative ability checking as well as compatibility checking presented in Section 5.1. It is able to test if the components can safely interoperate in the system when communicating according to their specified interaction rules. PIC also allows incremental testing of component protocol interoperability for subsystems, contributing to error discovery in system integration design at a lower cost.

At the time of writing, we have not yet incorporated any state space reduction techniques to reduce the checking complexity. This is left for future work.

## 7 Related work

The need for precise description of component interaction protocols to improve their interoperability has been widely recognised by the research community. A considerable number of formal language based approaches have been proposed in the literature to overcome the ambiguity of informal documentation used in the industry. For example, [7, 9, 22, 24] use finite state machines (FSMs), [2, 8] interface automata, [1, 4, 6] process algebra, [21] regular expressions, [3] Petri nets, and [5] description logics. The benefit of using a well-developed language is that many analysis

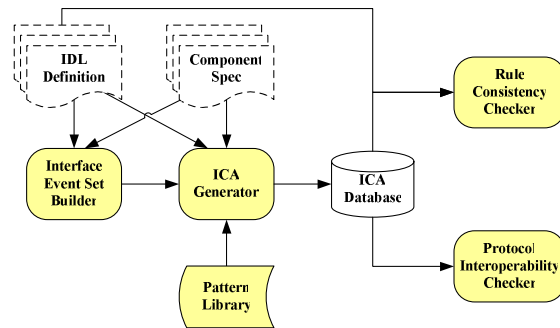


Figure 4. Main modules of the checking tool

techniques are already available. A major weakness, however, is that applying such an approach requires the user have sound knowledge of the underlying formalisms and methodologies. Their adoption is thus limited among practitioners who usually do not have the required expertise.

In contrast, our specification approach is designed for use by software practitioners. It is aimed at easing their burden in using mathematical tools to analyse components and systems. We approach this objective with two distinct means. First, we weigh usability over expressiveness and provide intuitive pattern operators for interaction rule specification, which hide the “intricate” formality from the user. None of the existing approaches provides interaction patterns to ease the specification. On the other hand, the semantic model developed in this paper builds on FSAs. It is as expressive as, if not more than, most of existing approaches. Based on this model, advanced users may extend our specification approach to gain more expressive power.

Second, we provide a simple way to support the incremental evolution of component interaction protocols, of which practitioners can have firm grasp. Unlike approaches that require the user encode all interaction rules of a component into a single protocol model, *e.g.* [3, 4, 6, 7, 8, 9, 21, 24], our work allows for relatively independent and declarative specification of these rules. This idea originates from [15, 16]. The work [5] follows a similar approach, but it uses description logics for rule specification and lacks a detailed study of the consistency of interaction rules. The idea of using multiple models to specify the component protocol was also present in [1, 2, 22]. For example, [1] uses port-based CSP process models, [2] uses one automaton for each pair of components, and [22] uses one call automaton for each component and one function automaton for each of its operations. Compared with those approaches, our interaction rules are more focussed and often easier to develop. The rules can be automatically combined to produce similar models to theirs. Further, none of those approaches explicitly addresses the consistency issues between multiple models of a component.

The issue of using component semantic specifications for system interoperability checking has been considered in [1, 2, 4, 6, 8, 14, 18, 21, 22, 24]. According to the employed composition mechanism, they can be grouped to two categories: rendezvous synchronization (including [1, 4, 6, 18, 21, 22, 24]) and asynchronous composition (including [2, 8]). Our approach adopts the second mechanism as we believe it resembles more closely the communication paradigm of possibly multi-threaded components in distributed systems. It also allows for more optimistic interoperability analysis for incomplete system designs. This can be a useful feature at the early design stages. Different from existing approaches, our approach to interoperability checking is able to provide feedback to assist in designing inter-component communication structures.

## 8 Conclusion

Designing interoperable component-based systems requires adequate semantic annotations for components as well as automated checking methods. In this paper, we have presented a lightweight pattern-based specification approach to component interaction rules and its semantic underpinning towards rule consistency and component interoperability. These, together with the proposed incremental checking approaches, would give software practitioners better access to mathematical means in analysing components and systems. We note, although a FSA-based semantics is developed to enable formal behavioural analysis, the user is not required to know FSAs to employ our approach.

Currently, we are investigating automated means to apply data abstraction techniques in building FSAs for interaction rules, in order to cater for data types from very large or infinite domains in consistency and interoperability checking. We also plan to apply this approach to more realistic systems and test its usability with practitioners.

**Acknowledgements.** We would like to thank Phan Manh Tan for his assistance in implementing the checking tool.

## References

- [1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3):213–249, 1997.
- [2] P. Attie and D. Lorenz. Establishing behavioral compatibility of software components without state explosion. Technical Report NU-CCIS-03-02, Northeastern University, 2003.
- [3] R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proc. European Conf. on Object-Oriented Programming*, pages 474–494, 1999.
- [4] M. Bernardo, P. Ciancarini, and L. Donatiello. Architecting families of software systems with process algebras. *ACM TOSEM*, 11(4):386–426, 2002.
- [5] A. Borgida and P. Devanbu. Adding more “DL” to IDL: Towards more knowledgeable component inter-operability. In [13], pages 378–387, 1999.
- [6] C. Canal, E. Pimentel, J. Troya, and A. Vallecillo. Extending CORBA interfaces with protocols. *The Computer Journal*, 44(5):448–462, 2001.
- [7] I. Cho. A framework for the specification and testing of the interoperation aspect of components. In [17], pages 53–64, 2000.
- [8] L. de Alfaro and T. Henzinger. Interface automata. In *Proc. Foundation of Software Engineering*, Softw. Eng. Notes 26(5), pages 109–120, 2001.
- [9] W. DePrince Jr. and C. Hofmeister. Enforcing a lips usage policy for CORBA components. In *EUROMICRO Conf. New Waves in System Architecture*, pages 53–60, 2003.
- [10] M. Dwyer, G. Avrunin, and J. Corbett. Specification patterns web site. <http://www.cis.ksu.edu/santos/spec-patterns>. Last accessed on 09 August 2005.
- [11] M. Dwyer, G. Avrunin, and J. Corbett. Property specification patterns for finite-state verification. In *Proc. Workshop on Formal Methods in Software Practice*, 1998.
- [12] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In [13], pages 411–420, 1999.
- [13] D. Garlan and J. Kramer, editors. *Proc. Int’l Conf. on Software Engineering*, 1999.
- [14] L. Grunske. Annotation of component specifications with modular analysis models for safety properties. In *Workshop on Component Engineering Methodology*, page 31, 2003.
- [15] J. Han. A comprehensive interface definition framework for software components. In *Proc. Asia-Pacific Software Engineering Conference*, pages 110–117, 1998.
- [16] J. Han and K. Ker. Ensuring compatible interactions within component-based software systems. In *Proc. Asia-Pacific Software Engineering Conference*, pages 436–445, 2003.
- [17] J. Hernández, A. Vallecillo, and J. Troya, editors. *Proc. ECOOP Workshop on Object Interoperability*, 2000.
- [18] P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. In *Proc. Fundamental Approaches to Software Engineering*, LNCS 2029, pages 60–75, 2001.
- [19] Y. Jin and J. Han. Runtime validation of behavioural contracts for component software. In *Proc. Int’l Conf. on Quality Software*, pages 177–184, 2005.
- [20] Y. Jin and J. Han. Specifying interaction constraints of software components for better understandability and interoperability. In *Proc. Int’l Conf. on COTS-based Software Systems*, LNCS 3412, pages 54–64, 2005.
- [21] F. Plasil and W. Visnovsky. Behaviour protocols for software components. *IEEE TSE*, 28(11):1056–1076, 2002.
- [22] R. Reussner. An enhanced model for component interfaces to support automatic and dynamic adaption. In [17], pages 33–42, 2000.
- [23] R. Smith, G. Avrunin, L. Clarke, and L. Osterweil. PROPEL: an approach supporting property elucidation. In *Proc. Int’l Conf. on Software Engineering*, pages 11–21, 2002.
- [24] D. Yellin and R. Strom. Protocol specifications and component adaptors. *ACM TOPLAS*, 19(2):292–333, 1997.