Abstract. Interface Definition Languages (IDLs) such as CORBA IDL/CIDL lack mechanisms for capturing semantic aspects of software components, such as their interaction protocols. This often poses significant problems for ensuring the behavioural interoperability between components in designing component-based systems. This situation is even more complicated where third-party components are involved. Our previous work in extending component interface specifications with interaction constraints presents a lightweight and incremental approach to capturing and validating the interoperability requirements of components. In this paper, we provide a deeper insight of this approach and further explore its capacity. We propose a formal semantic model for it, characterising the concepts of component interfaces and interaction constraints, the consistency between constraints as well as the behavioural interoperability between components. Based on this model, we demonstrate how interaction constraints of software components are formally defined without compromising the usability and understandability of this approach. We also illustrate how contradictory constraints and potential behavioural interoperability problems can be effectively detected.

1 Introduction

A key feature of component-based software engineering is that it allows the construction of an application using prefabricated pieces (or components) developed independently by third-parties so as to reduce costs and improve flexibility and reliability. In this construction process, it is essential to ensure that the individual components can in fact interoperate properly so as to achieve the desired functionality of the composite system. Typically, commercial Interface Definition Languages (IDLs) (including CORBA IDL/CIDL [12, 11]) are used to define the interfaces of components and expose the basic information to facilitate the use of components by third-parties. These IDLs, however, primarily address the signature aspects of software component interfaces, i.e., the names, parameters and data types of the provided operations. They do not provide support for capturing the semantic or behavioural aspects of a component, including its usage, capabilities and interaction behaviour. This often causes difficulties in enforcing the behavioural interoperability between components, especially between third-party components. That is, incorrect assumptions about the services of components often lead to incorrect usage, and therefore system malfunction.

To provide a sound support for studying component interoperability issues, unambiguous descriptions, clearly stating the semantics, qualities and usage/interaction protocol of a software component, are needed [6]. In particular, the usage/interaction protocol describes the rules that govern the interactions of the component, i.e., the valid
sequences of message exchanges or service/operations invocations. These rules define the way in which a component provides its services and the order in which its operations are to be invoked so as to facilitate the proper use of its services. There have been a number of proposals for the formal specification of component interaction protocols, such as [1–5, 13–15]. Most of these approaches employ formalisms with a strong mathematical flavour. This largely limits their use among software engineers or component developers who usually do not have the required expertise. Furthermore, they do not formally define the links between the signature aspects of component interfaces and the protocols. This makes it difficult to integrate these approaches with existing tools.

What is needed is a practical tool for protocol specification, which is not only intuitive and easy to use but also built on a sound formal foundation. In [9, 10], we have presented a lightweight specification approach based on interaction constraints. The approach uses intuitive temporal operators to define the interaction protocol of a component as a collection of interaction constraints. Each constraint presents an independently understandable piece of information about the protocol and states a sequencing or concurrency rule of selective operation invocations. As a result, this approach enables incremental protocol specification and realises a high usability and understandability.

In this paper, we provide this approach with a mathematical rigour and further explore its capacity in ensuring the interoperability between components. We develop a formal semantic model for constraint-based component interaction protocols. This model characterises the concepts of software component interfaces and interaction constraints, the consistency between constraints as well as the behavioural interoperability between components. It also provides a formal linkage between the signature and protocol aspects of component interfaces. We demonstrate how the formal semantics of interaction constraint specifications is rendered and how these constraints collectively infer a complete view of the component interaction protocol. We then illustrate with a case study the way to reason about constraint consistency and component behavioural interoperability using this semantic model. We also illustrate how to more effectively detect contradictory constraints and potential behavioural interoperation problems in a system, taking advantage of the independencies between interaction constraints.

The rest of this paper is organized as follows. Section 2 describes the background information. Section 3 presents the aforementioned semantic model. Section 4 extends our previous work with a formal semantics for interaction constraint specifications. Section 5 presents a case study on an auction system. Section 6 presents the related work. Finally, Section 7 contains the conclusions and future work.

2 Background

In this section, we first use an example to highlight the need for the precise specification of component interaction protocols. We then give an overview of our previous work in extending component interface specifications with interaction constraints.

2.1 The Need for Specifying Component Protocols

The example studied here is an auctioneer component, drawn and adapted from the distributed auction system in [4]. The auctioneer communicates with a number of bidders. It is able to accept registrations from the bidders and hold auctions among registered
bidders. It provides two operations for registration and unregistration and requires a bidder to provide three operations for interest inquiry, bidding settlement, and settlement announcement. Fig. 1 shows an example interface definition for the auctioneer, which can usually be described using a commercial IDL such as CORBA IDL/CIDL [11]. There, `ref No` stands for the reference number to a particular auction.

```plaintext
interface auctioneer {
  provides
  void register(in Bidder b);
  void unregister(in Bidder b);
  requires
  boolean wannaBid(in long ref No, in string itemDesc, in float price);
  void youGotIt(in long ref No, in float price);
  oneway void itemSold(in long ref No);
}
```

Fig. 1. An auctioneer interface definition with required operations

As shown, the interface definition does not convey semantic information about the auctioneer. In order for the system designer to deploy this component properly, additional information, e.g., its interoperability requirements, is needed. This may describe what services it provides, how it provides services, how the provided services can be utilised, and what are the obligations of the clients in using the services. For example, does the auctioneer only query registered bidders in an auction? Will an item be sold if there exists a bidding? Can an item be sold to a bidder who did not bid?

Without clear answers to the above questions, the system designer using the component may need to make certain assumptions. False assumptions may lead to malfunction of the system or even system failure. For instance, suppose the auctioneer can decide not to sell if there is only one bidding but bidders assume the otherwise. Then a bidder may wait forever when that happens. This may accumulatively cause system deadlock. Unfortunately, informal documentations attached with components cannot give a satisfactory solution due to their inherent ambiguity and inconsistency [10]. This highlights the need to devise sound principles and techniques for the component developer to explicitly and precisely specify the interoperability requirements of components, so as to assist the system designer in using the provided services in an appropriate manner as well as enforcing the interoperability between components in the system.

### 2.2 Extending Interface Specifications with Interaction Constraints

To solve the problems identified above, we presented in [9, 10] a lightweight and incremental approach to component interface specification. This approach extends the traditional interface definitions with protocol information which describes the rules that

---

1 Fig. 1 can be rewritten in CORBA CIDL using provided and used interfaces. Here we use a neutral representation to keep our approach independent of a particular middleware platform.
govern the interactions of a component with others and accordingly describes its interoperability requirements. In other words, the protocol specifies the sequencing or concurrency constraints between message exchanges and/or operation invocations. Such constraints are called interaction constraints.

Fig. 2 presents some example constraint specifications for the auctioneer component. Two kinds of constraints are distinguished: peer-level and interface-level. Peer-level constraints are to constrain the interactions with a particular neighbouring component, while interface-level constraints do not concern about the identities of the communicating neighbours but only the operation names and parameters.

```
interface auctioneer {
  ...
  peer-constraint
    wannaBid between register and unregister;
    youGotIt(ref No, *) causedby wannaBid(ref No, *, *, true);
  constraint
    wannaBid(ref No, *, *, true) leadsto itemSold(ref No);
}
```

Fig. 2. An Auctioneer Interface Definition with Interaction Constraints

As shown, each constraint applies a temporal operator to a number of invocation templates, where an invocation template specifies a group of related operation invocations and the operator states the sequencing relationships between the invocations. More specifically, an invocation template includes an operation and a partial evaluation to its parameters and return result. For descriptive convenience, we append the return value, if any, to the parameter list. A name in the parameter list represents that the constraint concerns the value of the corresponding parameter or return result, while the symbol "*" represents the otherwise. One may consider such a name represents a local variable and hence a constraint specification represents a list of concrete constraints, each assigning the variable a unique value. For brevity, we refer to an operation with only "*" in its parameter list by its name. For example, we write wannaBid for wannaBid(*, *, *, *).

In Fig. 2, two peer-level constraints and one interface-level constraint are specified. The first peer-level constraint applies to each individual bidder and ensures the relative sequencing among invocations to register, wannaBid and unregister from/to the bidder, regardless of the parameter values. Essentially, it requires that the auctioneer only query registered bidders for their interests in a particular auction. Note that wannaBid may be invoked several times between the other two operations. The second peer-level constraint ensures that if the youGotIt operation of a bidder is invoked regarding an auction ref No, this must result from a bidding of the bidder itself for ref No (i.e., wannaBid returns true). That is, an item being auctioned only be sold to a bidder who bid for it. The third interface-level constraint concerns about the sequencing between wannaBid and itemSold, regardless of the identities of neighbours. This specifies that an item being auctioned will be sold if there exists a bidding for it.

The above constraints present a partial set of rules governing the interactions of the auctioneer. It is, however, easy to add more constraints so as to make its interaction
behaviour more predictable. For instance, the auctioneer is required not to sell an item to two or more bidders at an auction. One can add an interface-level constraint stating that invocations \texttt{youGotIt(ref No, *)} can occur only once, if one ever occurs. One can also require that the auctioneer only invoke the operations \texttt{youGotIt} and \texttt{itemSold} of registered bidders, by specifying a peer-level constraint similar to the first one. For brevity, the details on how to state these constraints are omitted. Interested readers are referred to [9].

With the ability to easily add and remove constraints, the component developer is given the freedom to determine the extent to which the component’s interaction logic is made available to the user (or system designer) in order to facilitate its proper use while protecting its proprietary implementation techniques.

3 A Semantic Model for Component Interaction and Interoperability

As noted earlier, an interaction constraint represents a partial view on the interaction protocol of a component, and a set of constraints collectively infer a complete view on the protocol. This conforms well to the usual process of understanding, that is, comprehending the whole from the parts. On the other hand, we need to ensure that the collective inference produces a consistent view of the component interaction. This is an issue of detecting potential conflicts between constraints and ensuring their consistency. Furthermore, when designing a component-based system, we need to make sure that all the components interact in a way that their interaction constraints are not violated. This is an issue of enforcing the behavioural interoperability between the components.

To provide a solid basis for resolving these consistency and interoperability issues, we propose a formal semantic model for component interaction in this section. This model presents a formal definition of interaction constraints and enables the formal inference from these constraints to protocols and interoperability checking between components. It also formally define the links between interface signatures and interaction protocols. In Section 4, we shall use this model to define the semantics of interaction constraint specifications as shown above.

Below, we first define in Section 3.1 reactive transition systems and their composition for capturing the dynamic semantics of interaction constraints. We then present in Section 3.2 the semantic model, characterising interfaces, interaction constraints, and interaction protocols of software components. Based on this model, we define the protocol consistency in Section 3.3 and the component behavioural interoperability in Section 3.4.

3.1 Preliminaries

\textbf{Definition 1 (RTSs).} A reactive transition system (RTS) \( L \) is a tuple \((s^i, S, S^t, \Sigma, \Delta)\), where \( S \) is a set of states, \( s^i \in S \) is the initial state, \( S^t \subseteq S \) is a set of transient states, \( \Sigma \) consists of three mutually disjoint sets of input events \( \Sigma^i \), output events \( \Sigma^o \) and internal events \( \Sigma^h \), and \( \Delta \subseteq S \times \Sigma \times S \) is a set of steps. A RTS is deterministic if \( \forall e \in \Sigma, s, s', s'' \in S, (s, e, s'), (s, e, s'') \in \Delta \) implies \( s' = s'' \).

Transient states capture a liveness requirement that a RTS cannot stay at a transient state without making further progress. They are used later to detect both component
internal deadlocks and system deadlocks. Input, output and internal events are distinguished to reflect the fact that a software component has control over its output and internal events but no control over its input events. The occurrences of input events are rather controlled by its environment. Throughout this paper, we assume that any internal event of a RTS is unique to the RTS and is not an event for any other RTS. In the following, we shall only consider deterministic RTSs. The definition below formulates the operational semantics of RTSs in terms of traces and languages.

**Definition 2 (traces, languages).** A trace of a RTS $L$ is an event sequence $e_1e_2 \ldots e_n$ such that $\exists s_1, \ldots, s_n \in S, \langle s_i, e_i, s_i' \rangle \in \Delta \cap \forall j \in \mathbb{N}: 2 \leq j \leq n, \langle s_{j-1}, e_j, s_j \rangle \in \Delta$. A finite trace is said to be acceptable by $L$ if $s_n \notin S^i$. An infinite trace (where $n = \infty$) is acceptable by $L$ if for any $k \in \mathbb{N}$, there always exists $j \in \mathbb{N}: j > k, s_j \notin S^i$. The set of all acceptable traces of $L$ is called the language of $L$.

For the purpose of deadlock detection, we exclude from the language of a RTS both finite traces ending at a transient state and infinite traces looping only between transient states. The composition of RTSs is defined as follows.

**Definition 3 (composition).** Consider two RTSs $L_1$ and $L_2$. Let $\Sigma^{syn} = \Sigma_1 \cap \Sigma_2$, then the composition of $L_1$ and $L_2$ is defined as a RTS $L = (s', S, S', \Sigma, \Delta)$ such that

- $s' = \langle s_1', s_2' \rangle$, $S \subseteq S_1 \times S_2$, and $S' \subseteq (S_1' \times S_2') \cup (S_1 \times S_2')$;
- $\Sigma = \Sigma_1 \cup \Sigma_2$;
- For all $\langle (s_1, s_2), e, (s_1', s_2') \rangle \in \Delta$, the following conditions hold:
  - If $e \in \Sigma^{syn}$, then $\langle s_1, e, s_1' \rangle \in \Delta_1 \land \langle s_2, e, s_2' \rangle \in \Delta_2$;
  - If $e \in \Sigma_1 \setminus \Sigma^{syn}$, then $\langle s_1, e, s_1' \rangle \in \Delta_1 \land s_2 = s_2'$;
  - If $e \in \Sigma_2 \setminus \Sigma^{syn}$, then $s_1 = s_1' \land \langle s_2, e, s_2' \rangle \in \Delta_2$.

From the above, any constituent RTS being at a transient state means the whole composition being a transient state. If we further compose the composed RTS with a RTS $L_3$, we can have the composition of $L_1$, $L_2$ and $L_3$. Generalising this will give us the composition for a number of components.

With no distinction between input and output events, Definition 3 presents a general composition scheme. To make the distinction, we define two specialised schemes: unification and synchronisation. Unification is intended for computing the language intersection of RTSs by combining the same type of events. This scheme is later used for deducing its interaction protocol of a component from its interaction constraints. In contrast, synchronisation concerns about the communication and compatibility of RTSs and binds output events of a RTS to input events of another RTS. This scheme is later used for defining the compatibility and interoperability between components.

**Definition 4 (unification).** Consider two RTSs $L_1$ and $L_2$ such that $\Sigma_1' \cap \Sigma_2'' = \emptyset$ and $\Sigma_1'' \cap \Sigma_2' = \emptyset$. Let $L = (s', S, S', \Sigma, \Delta)$ be their composition (Definition 3). Then $L$ is called the composition by unification if $\Sigma' = \Sigma_1' \cup \Sigma_2', \Sigma'' = \Sigma_1'' \cup \Sigma_2'$, and $\Sigma^h = \Sigma_1^h \cup \Sigma_2^h$.

The unification scheme is only applied to RTSs where any input event of any RTS is not an output event of any other RTS. The input, output and internal events of the composition are the union of those of the constituent RTSs, respectively. Definition 4 basically requires that an acceptable trace of the composition be acceptable for all the constituent RTSs, disregarding the events unknown to them. Next, to define the synchronisation scheme, we need a preliminary definition (given below) on RTS compatibility.
Consider two RTSs $L_1$ and $L_2$ such that $\Sigma_1 \cap \Sigma_2 = \emptyset$ and $\Sigma_1 \cap \Sigma_2 = \emptyset$. Then $L_1$ and $L_2$ are called compatible if there exists a relation $\sim \in S_1 \times S_2$ such that (1) $s_1^1 \sim s_2^1$ and (2) for any $s_1 \in S_1$, $s_2 \in S_2$: $s_1 \sim s_2$, the following conditions hold:

- If $\exists e \in \Sigma_1 \setminus \Sigma_2$: $(s_1, e, s_1') \in \Delta_1$, then $s_1' \sim s_2$;
- If $\exists e \in \Sigma_2 \setminus \Sigma_1$: $(s_2, e, s_2') \in \Delta_2$, then $s_1 \sim s_2'$;
- If $\exists e \in \Sigma_1 \cap \Sigma_2$: $(s_1, e, s_1') \in \Delta_1$, then $\exists (s_2, e, s_2') \in \Delta_2$: $s_1' \sim s_2'$;
- If $\exists e \in \Sigma_2 \cap \Sigma_1$: $(s_2, e, s_2') \in \Delta_2$, then $\exists (s_1, e, s_1') \in \Delta_1$: $s_1' \sim s_2$.

The compatibility is only related to RTSs with mutually disjoint sets of input/output events. The compatibility means that a RTS is always able to consume an input event, whenever the other RTS produces this event as its output. The composition of two RTSs by synchronisation is then defined as follows.

Consider two compatible RTSs $L_1$ and $L_2$ as in Definition 5. Let $\Sigma^{syn} = \Sigma_1 \cap \Sigma_2$, $L = (s^i, S^i, S^o, \Sigma^i, \Delta)$ be their composition (Definition 3). Then $L$ is called the composition by synchronisation if $\Sigma^i = (\Sigma_1 \cup \Sigma_2) \setminus \Sigma^{syn}$, $\Sigma^o = (\Sigma_1 \cup \Sigma_2) \setminus \Sigma^{syn}$, and $\Sigma^h = \Sigma_1^h \cup \Sigma_2^h \cup \Sigma^{syn}$.

This scheme internalises the events shared between the constituent RTSs in the composition. Hence these events will no longer participate in further synchronisation.

### 3.2 Characterizing Interfaces of Software Components

To employ RTSs to capture the interaction protocol of software components, we need to know what an event is for a component and how events are related to the component interface. To do so, we first characterise an interface by a number of message types, operations and exceptions, and then define events for these elements and for the interface. In doing so, we assume a universe of values $V$.

A message type specifies a group of messages that a component may receive and produce. Unlike the messages corresponding to operation invocations described later, there is generally no built-in causality relation between these messages. In some middleware platforms such as CORBA, such message types correspond to one-way operations, typically used to capture asynchronous communications between components. Formally, message types and messages are defined as follows.

**Definition 7 (message types, messages).** A message type is defined by $mt = (id, \rho, \theta)$, where $id$ is the type name, $\rho$ is a set of parameter names, and $\theta: \rho \rightarrow 2^V \setminus \emptyset$ is a total type function mapping each parameter name to a non-empty subset of values from the universe. A message of type $mt$ is a tuple $(id, m)$, where $m$ is a total function $\{p \rightarrow v \mid p \in \rho, v \in \theta(p)\}$. The set of all messages of type $mt$ is denoted by $\mathcal{M}_{mt}$.

A message type identifies a name and a set of typed parameters. A message is a total evaluation of all the parameters of its type. In contrast to message types, an operation specifies a unit of synchronous communication. As defined below, it is composed of a name, a list of input and output typed parameters, a return type, and exceptions.

**Definition 8 (operations).** An operation is defined by $op = (id, \rho^i, \theta^i, \rho^o, \theta^o, \chi)$, where $id$ is the operation name, $\rho^i$ and $\rho^o$ are sets of input and output parameter names, respectively, $mt^i = (id, \rho^i, \theta^i)$ and $mt^o = (id, \rho^o, \theta^o)$ are both message types, and $\chi$ is a set of message types for exceptions.
For some operation, we may have $\rho^i \cap \rho^o \neq \emptyset$. In this case, the elements in $\rho^i \cap \rho^o$ are both input and output parameters. For operations with return types, we assume there always is a keyword “result” in $\rho^o$ and $\theta^o$ (“result”) gives the return result. For an operation, $\chi$ consists of all the message types of exceptions that the operation can raise. In the following definition, we associate two kinds of messages to an operation.

**Definition 9 (call, reply messages).** Given an operation $op$, a call message of $op$ is a message of type $mt^i$. A reply message of $op$ is a tuple $(id, cm, rm)$ such that $(id, cm)$ is a call message of $op$ and $(id, rm)$ is a message of type $mt^o$.

A call message represents an invocation to the operation. A reply message always results from the completion of a specific invocation. We make this causality explicit by letting a reply message be a pair consisting of a message of type $mt^o$ and the call message that causes it. In the following, we let $M^c_{op}$ and $M^r_{op}$ denote the set of all call and reply messages of $op$, respectively. We also let $M_{op} = M^c_{op} \cup M^r_{op}$.

As shown in Fig. 2, an interaction constraint specification allows the user to abstract away irrelevant parameters using symbol “∗”. A resultant constraint will concern the sequencing between sets of messages rather than individual messages. A message in such a set can assign an arbitrary value to an irrelevant parameter. To facilitate the study of such constraints, we define message sets using partial instantiation functions.

**Definition 10 (message sets).** Given a message type $mt = (id, p, \emptyset)$ and a (partial) function $f: \rho \rightarrow \mathbb{V}$, the set of messages related by $f$ is defined by $M_{mt|f} = \{id\} \times \{m: \rho \rightarrow \mathbb{V} \mid \forall p \in \rho, m(p) \in \theta(p), (p, m(p)) \in f \lor \not\exists (p, v) \in f\}$. We call $f$ an instantiation function for $mt$.

Clearly, $M_{mt|f} \subseteq M_{mt}$ and $M_{mt|\emptyset} = M_{mt}$. Now consider an operation $op$, instantiation functions $f^{i}$ (for $mt^{i}$) and $f^{o}$ (for $mt^{o}$). We let $M^{c}_{op|f^{i}} = M^{r}_{mt^{i}}$ be the set of call messages of $op$ related by $f^{i}$, $M^{r}_{op|f^{i}, f^{o}} = M^{c}_{mt^{i}} \times M^{r}_{mt^{o}}$ be the set of reply messages of $op$ related by $(f^{i}, f^{o})$, and $M_{op|f^{i}, f^{o}} = M^{c}_{op|f^{i}} \cup M^{r}_{op|f^{i}, f^{o}}$.

**Definition 11 (component interfaces).** Given a component, its interface is a tuple $if = (MT, OP, XT)$, where $MT = MT^{i} \cup MT^{o}$ is a set of input and output message types, $OP = OP^{prov} \cup OP^{req}$ is a set of provided and required operations, $XT = XT^{i} \cup XT^{o}$ is a set of message types of input and output exceptions, and $MT^{i}, MT^{o}, OP^{prov}, OP^{req}, XT^{i}$ and $XT^{o}$ are mutually disjoint.

Message types in $MT$ are used for asynchronous communication or one-way operation invocations, while operations in $OP$ are for synchronous communication. Also, $XT$ specify exceptions that can be raised or handled in operations. For simplicity, we assume mutual exclusion between $XT^{i}$ and $XT^{o}$, i.e., an exception type is used for either exception production or consumption by the component but not both. Also, we require that $\chi_{op} \subseteq XT$ hold for any operation $op$ in $OP$.

### 3.3 Formalising Interaction Constraints and Protocol Consistency

We have so far characterised the syntactic aspects of component interfaces. In this section, we consider interaction protocols. We first define interface events, interaction constraints by RTSs, and then their protocol and consistency by RTS unification.
Definition 12 (interface events). Given a component with interface \(i\) and a set of its neighbouring components \(N\), the set of interface events of the component is defined by \(\Sigma^i = N \times (\bigcup_{mt \in MT} M_{mt} \cup \bigcup_{op \in OP} M_{op} \cup \bigcup_{xt \in XT} M_{xt})\).

An interface event identifies a neighbouring component and a message. The message can be an input, output, call, reply or exception message of the component. Accordingly, the neighbour may be the event producer or consumer depending on the type of the message. To make this explicit, we partition \(\Sigma^i\) into two disjoint sets \(\Sigma^i\) and \(\Sigma^o\) such that \(\Sigma^i\) contains all the elements involving an input, call, reply or exception message from a neighbour, and \(\Sigma^o\) consists of all the elements involving an output, call, reply or exception message to a neighbour. We also let \(\Sigma^x = N \times \bigcup_{xt \in XT} M_{xt}\). The following definition defines interaction constraints as deterministic RTSs.

Definition 13 (interaction constraints). Let \(i\), \(N\) and \(\Sigma^i\) be as above, then an interact constraint is defined by a deterministic RTS \((s^i, S, S^t, \Sigma^i, \Delta)\).

The following definition states how the interaction protocol is deduced from a given set of interaction constraints and how the consistency between constraints can be ensured. Basically, the interaction protocol is a RTS composed by unification of constraints and the consistency refers to the non-emptiness of the protocol language.

Definition 14 (interaction protocols, protocol consistency). Let \(i\) and \(N\) be as above and \(IC\) a set of interaction constraints, the interaction protocol of the component is defined by the composition of all constraints in \(IC\) by unification. The constraints in \(IC\) are said to be protocol consistent if the language of the interaction protocol is not empty.

3.4 Formalising Component Interoperability

Based on the interaction protocols of components, we define the behavioural interoperability between them at two levels: pair-wise and system-wide. The pair-wise interoperability concerns the mutual respect of interaction constraints between a pair of neighbouring components. It requires the protocols of the components to be compatible. The system-wide interoperability ensures the non-violation of interaction constraints of all components in the system. It requires the language non-emptiness of the RTS composed by synchronisation from the interaction protocols of these components. Note that we assume all data type mismatches between components have been ruled out.

Definition 15 (component interoperability). Given two components \(c_1, c_2\) and their interaction protocol RTSs \(L_1\) and \(L_2\), \(c_1\) and \(c_2\) are called interoperable if \(L_1\) and \(L_2\) are compatible. Given a system with components \(c_1, \ldots, c_n\), the components are called interoperable if they are mutually interoperable and the language of composition of their interaction protocols by synchronisation is not empty.

As defined, the checking of interoperability at both levels can be computationally expensive due to the potential state space explosion problem in the process of computing the interaction protocols from constraints and composing the protocols. The size of state space of a composed RTS, e.g., a protocol, may grow exponentially with the number of the constituent RTSs, e.g., constraints. However, we observe in Section 2.2 that different interaction constraints may restrict on different sets of events. For example,
peer-level constraints applied to different neighbours are mutually independent. This means that, in order to check the component interoperability, it may be unnecessary to compute the whole protocols. Rather, composing the constraints restricting on a common set of events is sufficient to determine the interoperability. This also applies to the consistency checking between constraints. While a formal treatment to this issue is beyond the scope of this paper, we shall make use of this observation in Section 5 and illustrate its usefulness for detecting contradictory interaction constraints and potential component interoperability problems. Such detection is of practical value, as it is often the case that detecting problems is much more important than ensuring the absence of problems in the development of real-world applications.

4 Formalising Interaction Constraint Specifications

We have introduced a semantic model for component interaction. In this section, we build on this model and define the formal semantics of interaction constraints specifications as shown in Section 2.2, so as to enable sound consistency and interoperability reasoning for specified software components. In the following, we first map invocation templates in these constraint specifications to event sets, and then define temporal operators in terms of RTSs parameterized by the event sets.

4.1 Mapping Invocation Templates to Event Sets

To formalise invocation templates, we not only need to consider its constituents but also the variables used, the variable correspondence between the templates and the level of the enclosing constraints (in other words, the identities of neighbouring components).

As shown in Fig. 2, an invocation template takes the form of $id(v_1, \ldots, v_n)$ or $id(v_1, \ldots, v_n, \gamma)$, where $id$ refers to a message type or operation, $v_1, \ldots, v_n$ are variables or values representing the arguments for each formal parameter of $id$, and $\gamma$ is a variable or value representing the return result of an operation invocation. Semantically, a template where $id$ refers to a message type (or one-way operation) $mt$ can be represented as a tuple $(mt, f_i)$ such that $f_i$ maps every input parameter name at position $j$ to $v_j$ except that $v_j = "\ast"$. Likewise, a template where $id$ refers to an operation $op$ is represented by a tuple $(op, f_i, f_o)$ such that $f_i$ is as above and $f_o$ maps every output parameter name at position $j$ to $v_j$ unless $v_j = "\ast"$ and maps the return result, say "result", to $\gamma$ unless $\gamma = "\ast"$ or the result is absent. For descriptive convenience, we often write $(mt, f_i, \emptyset)$ instead of $(mt, f_i)$. As an example, "youGotIt(refNo, \ast) causedby wannaBid(refNo, \ast, \ast, true)" uses a variable $refNo$ and corresponds to $(wannaBid, f^i, f^o)$ such that $f^i = \{"refNo" \mapsto refNo\}$ and $f^o = \{"result" \mapsto true\}$. Essentially, it represents a set of invocations and replies to wannaBid, of which the call messages are bidding inquiries concerning auction $refNo$ and the reply messages give a truth value to the inquiries.

As noted, a constraint may use local variables to state the value correspondence between invocation templates. Such a constraint is basically a shorthand for a list of concrete constraints, each assigning the variables a valid value. For instance, in Fig. 2, "youGotIt(refNo, \ast) causedby wannaBid(refNo, \ast, \ast, true)" can be interpreted as "youGotIt(l, \ast) causedby wannaBid(l, \ast, \ast, true)" for all $l \in long$.

In Section 2.2, we made a distinction between peer-level and interface-level constraints. A peer-level constraint restricts the sequences of operation invocations with a
specific neighbouring component, while a interface-level constraint disregards the identities of communicating neighbours. To map invocation templates to event sets, we also need to resolve the identities of neighbouring components being constrained. For example, in Fig. 2, "youGotIt(refNo, ∗) caused by wannaBid(refNo, ∗, ∗, true)" is a peer-level constraint. Semantically, it states that "\{(b), (youGotIt, f), (\emptyset)\} caused by ((\{b\}, (wannaBid, f', f'')))" must hold for any bidder b in the system, where f' and f'' are as above. In addition, "wannaBid(refNo, ∗, ∗, true) leads to itemSold(refNo)" is an interface-level constraint. Suppose B is the set of all bidders in the system, then this constraint ensures "(B, (wannaBid, f', f'')) leads to (B, (itemSold, f, \emptyset))".

As shown above, an invocation template in an interaction constraint specification can formally be represented by: a set of neighbours N, an operation op or a message type mt, and two instantiation functions f' and f", where f" \equiv \emptyset for a message type mt. For a message type mt, we call tuple mi = (N, mt, f', \emptyset) an message invocation template (MIT) and assign to it a set of interface events: \(\Sigma_{mi} = N \times M_{mt} \setminus f\). Each element in this set is a pair composed of a neighbour in N and a message of type mt. In addition, for an operation op, we call tuple oi = (N, op, f', f") an operation invocation template (OIT) and assign to it a set of interface events: \(\Sigma_{oi} = N \times M_{op} \setminus (f_1, f_2)\). Since operation invocations involve two kinds of events, we obtain two subsets: call events \(\Sigma^c_{oi} = N \times M^c_{op} \setminus f\), and return events \(\Sigma^r_{oi} = N \times M^r_{op} \setminus (f_1, f_2)\). The former set consists of all the call messages, while the latter consists of all the reply messages.

It should be noted that even though an invocation template for an operation corresponds to two sets of interface events, we do not require the user to distinguish between them in interaction constraint specifications. This is to avoid burdening the user (or component developers) but provide a lightweight specification approach. As shown below, we rather define temporal operators to make the distinction.

4.2 Mapping Temporal Operators to Parameterized RTSs

Temporal operators are introduced for describing recurring patterns of temporal or sequencing relationships between interface events of a component. In this section, we demonstrate the approach to defining their semantics in terms of RTSs parameterized by event sets, using the three operators in the auctioneer example. The operators include caused by, leads to and between. Other operators, however, can be found in [9]. Below, we assume \(E_1, E_2, E_3 \subset \Sigma \setminus \Sigma^x\) are mutually disjoint sets of interface events (\(\Sigma^x\) is the set of exception events), \(mi_1, mi_2, mi_3\) are MITs, and \(oi_1, oi_2, oi_3\) are OITs.

Constraint statement "\(E_2\) caused by \(E_1\)" indicates the causality between events in \(E_1\) and \(E_2\). More specifically, whenever an event in \(E_2\) occurs, an event in \(E_1\) must have occurred. The semantics of this statement is described in a compact form by the reactive transition system shown in Fig. 3, where grey circles represent states, the circle pointed to by an arrow with no source represents the initial state, and arcs between states represent steps. Also, \(O = \Sigma \setminus (E_1 \cup E_2 \cup \Sigma^x)\) is the set of all other non-exception events. The RTS transits between states when any event in a labelling set of a step occurs. The absence of \(E_2\) at the initial state implies that any event occurrence in \(E_2\) at that state will violate the specified constraint. For invocation templates, we let "\(mi_2\) caused by \(mi_1\) \(\triangleq\) \(\Sigma_{mi_2}\) caused by \(\Sigma_{mi_1}\)" and "\(oi_2\) caused by \(oi_1\) \(\triangleq\) \(\Sigma_{oi_2}\) caused by \(\Sigma_{oi_1}\)".

Constraint statement "\(E_1\) leads to \(E_2\)" dictates the eventuality of event occurrences in \(E_2\) as a result of an event occurrence in \(E_1\). That is, after any event occurrence in \(E_1\), there will eventually be an event occurrence in \(E_2\) (before the component terminates).
This is stronger than the “causedby” statement which only indicates the possibility of event occurrences in $E_2$. The RTS semantics of this statement is depicted by Fig. 4, where $\ominus$ denotes a transient state and $O$ is as above. As noted, a transient state excludes any trace ending at it from the RTS language. An event $E_2$ needs to occur at the lower state in order for the RTS to move into the upper regular state. For invocation templates, we let “$mi_1 \leadsto mi_2 \triangleq \Sigma_{mi_1} \leadsto \Sigma_{mi_2}$” and “$oi_1 \leadsto oi_2 \triangleq \Sigma_{oi_1} \leadsto \Sigma_{oi_2}$”.

Constraint statement “$E_2$ between $E_1$ and $E_3$” restricts the event occurrences in $E_2$ relative to $E_1$ and $E_3$, where events in $E_1$ are the “on” switch and events in $E_3$ are the “off” switch. More specifically, any event occurrence in $E_2$ is possible only when an event in $E_1$ has occurred and no event in $E_3$ has occurred afterwards. Fig. 5 presents its corresponding RTS, where $\text{O} = \Sigma_{E_1 \cup E_2 \cup E_3 \cup \Sigma^r}$. As shown, events in $E_2$ can occur only when the RTS is at the lower state. For invocation templates, we let “$mi_2 \text{ between } mi_1 \text{ and } mi_3 \triangleq \Sigma_{mi_2} \text{ between } \Sigma_{mi_1} \text{ and } \Sigma_{mi_3}$” and “$oi_2 \text{ between } oi_1 \text{ and } oi_3 \triangleq \Sigma_{oi_2} \text{ between } \Sigma_{oi_1} \text{ and } \Sigma_{oi_3}$”.

It is worth noting that, in defining these temporal operators, we have taken a biased view on exception events, excluding their occurrences from the interaction protocol of a component. However, other alternatives can be accommodated by changing the elements in the labelling sets of RTS steps.

5 A Case Study

We have demonstrated the use of our semantic model in defining the semantics of interaction constraint specifications as given in Section 2.2. In this section, we illustrate with an auction system the key techniques to reason about the consistency between constraints and the interoperability between components. Suppose an auction system consists of an auctioneer $a$ and three bidders $b_1, b_2, b_3$, and the auctioneer will auction two items 1 and 2. We let $B = \{b_1, b_2, b_3\}$, $f^a(x) = \{\text{refNo} \mapsto x\}$, and $f^o = \{\text{"result"} \mapsto \text{true}\}$.

To reason about the constraint consistency of the auctioneer, we build for the constraint specifications in Fig. 2 their semantic representations, each assigning variable $\text{refNo}$ a unique item number. We also build for each bidder a semantic representation of a peer-level constraint. The resultant constraints are shown in Fig. 6. We then construct a RTS for each semantic constraint and compute their composition by unification. As the language of the composed RTS is not empty, we know from Definition 14 that these constraints are protocol consistent.

Next, suppose the constraints for bidders are specified by Fig. 7, where “this” refers to a bidder itself and the “@pre” statement defines a precondition for all $\text{register}$ in-
vocations, which requires that no bidder register on the behalf of another. As above, we can ensure the consistency of these constraints. We then need to check the interoperability between the auctioneer and the bidders in building the auction system. To do so, we may construct the protocol RTS for each component, compute their composition by synchronisation, and then determine the interoperability. As noted in Section 3.4, this checking method may involve a high computational complexity. In cases where the interoperability does not hold, it can be very difficult to work out the cause of the problem. Therefore, we illustrate below with the auction system an effective approach to detecting the interoperability problems and identifying the cause. This approach can also be used to detect contradictive constraints.

The interoperability does not hold in the example auction system. To find out the cause, we examine the constraints for bidders one by one. Consider the first constraint. As the same constraint also appear in the auctioneer specification and is proved there to be consistent with the others, this constraint clearly does not cause an interoperability problem. As register is a required operation of a bidder, the second constraint only restricts output events of a bidder itself. Also, testing its compatibility with each constraint of the auctioneer does not yield negative results. Hence, the second constraint is not the cause for the interoperability problem. The third constraint states that a settlement notification youGotIt will eventually be received as a result of each bidding (i.e., wannaBid with a truth return value). This dictates a stronger expectation than the auctioneer can promise. Composing its RTS with the second constraint of the auctioneer will easily identify that it causes the interoperability problem.

Now consider two generic components both with internally consistent interaction constraints. To detect potential pair-wise interoperability problems between them, we first group constraints according to the interface events being constrained and compute the composition of constraints (called a mini-protocol below) in each group. For exam-
ple, the events constrained in the first constraint in Fig. 7 are both the call and return events of `wannaBid`, and those in the second constraint are the call events. We can thus group these two constraints together. We then compare each pair of related mini-protocols from the two components. Two mini-protocols are said to be related if the constrained interface events are related. If a pair of mini-protocols are not compatible, then a source for interoperability problems is identified. Although more involved, a similar approach may be taken to detect system-wide interoperability problems. A formal treatment to these issues is left for future work.

6 Related Work

There exists a body of work in the formal specification of protocol information in component interfaces, e.g., [1–5, 13–15]. Most of them assume that the user have an expertise or sound knowledge in formal languages, e.g., finite state machines (FSMs) [5, 14, 15], regular expressions [13], process algebra [1, 4], Petri nets [2], or description logics [3]. On the contrary, our specification approach assumes little of such knowledge from the user [10]. In this work, we still maintain its advantages in the understandability and usability required for practical use. We also develop a sound semantic foundation for it and provide the preciseness required for automated reasoning. Furthermore, we formally define the links between the syntactic aspects of component interfaces and the component interaction protocols. These links are often missing or informally described in other approaches. This definition facilitates the integration of our approach with existing tools that support IDLs.

In particular, our approach differs from [5–7, 14, 15] in that these approaches assume atomic executions of operation invocations, which limits their ability to capture the relationships between concurrent invocations; it differs from [1, 4, 13] in its ability to support incremental protocol specification and runtime validation of partial interaction protocols [7]; it differs from [2] in its ability to hide the internal semantics of components from their external behaviour. A more detailed comparison with these approaches can be found in [7].

7 Conclusion

The proper use of software components in a distributed system is critical to the correct functioning of the system. This is especially the case when third-party components are involved. To ensure the proper use, unambiguous specifications of component interaction protocols are needed. To meet this need, we have extended in this paper the work in [9, 10] with a formal semantic model. This model characterises the concepts of component interfaces and interface constraints as well as constraint consistency and component interoperability. This enables us to utilise the preciseness of formal methods to support automated reasoning about component interoperability and the validation of runtime component interaction. On the other hand, we maintain the usability and understandability by hiding the semantic model behind intuitive temporal operators.

Based on the semantic model, we have formally defined the semantics of interaction constraint specifications in terms of RTSs. This enables reasoning about the protocol compatibility as well as the deadlock freedom in component interaction. Also, we have
demonstrated the way to formally deduce the interaction protocol of a component from its interaction constraints, and the way to check the consistency between the constraints and the behavioural interoperability between components.

Currently, we are investigating the development of tool support for automatic consistency checking of component interaction constraints as well as design-time interoperability checking. We are also extending a validation tool called RIDLMON [7] for validating runtime inter-component communications against the components’ interaction constraints.

References