

# Rich Interface Specification for Software Components

Jun Han

Peninsula School of Computing and Information Technology  
Monash University, McMahons Road, Frankston, Vic 3199, Australia  
phone: +61 3 99044604, fax: +61 3 99044124, e-mail: jhan@monash.edu.au

**Abstract.** Current industrial models for software components have made component-based software engineering a practical reality. However, these models are limited in the sense that their support for the specification of component interfaces primarily deals with syntactic issues. As such, component mismatch remains a critical stumbling block in component based software engineering: components may syntactically “plug”, but still not behaviourally “play”. To facilitate proper understanding and use of components and advance towards full “plug-and-play”, richer interface specification is needed for components. In this article, we introduce a framework for rich component interface specification. It addresses the issues of interface signature (syntax), interface configurations (structure), interface behaviour (semantics), interaction protocols (constraints) and quality properties of software components. In addition to the framework as a whole, two specific contributions are (1) its use of interface configurations in capturing and organising usage-specific characteristics, and (2) its constraint-based approach to the incremental specification of interaction protocols. A number of applications are also highlighted in the article to illustrate the framework’s the main objective of immediate practical usability.

**Keywords:** Software components, interface description languages, interface signature, interface configurations, behavioural semantics, interaction protocols, quality properties, formal constraints, “light-weight” semantics.

## 1 Introduction

Software systems form an essential part of most enterprises’ business infrastructure, and become increasingly complex. In today’s global market, these enterprises have to adjust and improve their business practices on an on-going basis to gain and maintain a competitive edge. Such changes to business practices often raise requirements for change to their underlying software systems and the need for new systems, which have to be fulfilled in a timely fashion. It is in this business context that being able to assemble or adapt software systems with reusable components proves to be vital.

We have seen examples of integrating software components or packages into existing systems to achieve specific business objectives of an enterprise. Perhaps, the most prominent is the use of Commercial-Off-The-Shelf (COTS) software packages in enterprise systems. Experience has shown that even with advanced technological support it is not, in general, an easy task to assemble software components into systems. A major issue of concern is the mismatches of the components in the context of an assembled system, especially when the mismatches are not easily identifiable [7]. The mismatches are largely due to the fact that the capability of the components are not clearly described or understood at the time of use.

Commercial software components or packages are usually delivered in binary form. Relevant documentation, such as the user’s manual, has to be relied on to understand their capability and

ways of use. In essence, such documentation forms the description of the software's external interface. However, such interface descriptions in natural languages do not provide the level of precision required for component understanding, and therefore have contributed to the above mentioned mismatches. When discovering and loading components into running systems dynamically, it becomes a must that the components have precise and even comprehensive interface descriptions.

Ideally one needs to look only at the interface specification of a component to understand how to use it. Interface description languages (IDLs) have been developed to facilitate reuse and inter-operation of distributed objects/components through interface descriptions. The most prominent are those of CORBA, COM/DCOM and Java/JavaBeans. These IDLs deal with primarily syntactic issues, i.e., the forms and types of the interface elements (namely, attributes, operations and events). With syntactically compatible interface descriptions, components may well "plug", but may still not "play" as their behaviour may be incompatible. To gain clear understanding of a component's *exact* capability and achieve better or full "plug-and-play", the interface description of the component should also include other essential aspects, including the usage-oriented organisation of interface elements, the behavioural semantics of interface elements, the interaction protocols regarding the elements' use, and the non-functional or quality properties of the component. This article is to address these issues and extend the current IDLs with capabilities for richer interface specification.

Traditionally, the interface signature of a component makes explicit the services provided by the component in the form of attributes, operations and events. When being used in a given context, in general, a component not only provides services to other components but also requires services from them and from still other components. As such, the component's interface signature should also explicitly specify the interface elements for required services [25]. In the given context, the component plays a number of roles relative to its neighbouring components from specific perspectives. To help the understanding and proper use of the component, it is beneficial to structure its interface elements into a role-based organisation, i.e., *an interface configuration*. Furthermore, a component may be used in different application contexts, and therefore may have multiple interface configurations.

Understanding the semantics of a component's interface elements is fundamental to understanding the component's behaviour, and therefore to its proper use. For example, we need to know whether there are further constraints on the value range of an attribute in addition to that dictated by its type. For an operation, we need to know its assumptions and effects. For an event, we need to know whether there are specific characteristics or requirements about its generation and use.

Using its interface elements, a component can be dynamically interacted with in numerous ways. At a given moment in time, certain interactions with the component are successful, while others generate exceptions, errors or unpredictable behaviours. The reason is that this latter class of interactions violate the assumptions of the relevant interface elements. While providing behavioural definition, the semantic specification of the individual interface elements does not provide *direct, obvious* guidance on how to use these elements in harmony to avoid the interactions that lead to exceptions, errors or unpredictable behaviours. Furthermore, in concurrent and distributed applications, assuming that the services provided by a component are atomic and are executed as transactions is not always practical or true [3]. Additional synchronisation constraints between the interface elements of a component may be required. Therefore, the synchronisation dependency between the interface elements needs to be specified in the interface as the *interaction protocols* or *interaction semantics* of the component, to facilitate its proper use.

The behaviour and relationships of a component's interface elements are mainly about the component's functionality. The non-functional or quality properties of the component are also of vital significance to its use. These quality properties concern the component's characteristics regarding performance, reliability, security, and so on. The availability of such properties' specification through the component's interface will help the user to assess the component's impact in a given

context of use and to decide its suitability.

The main contributions of this article are as follows:

- We provide a practical framework for rich component interface specification. It addresses the issues highlighted above, by incorporating existing techniques when possible and developing new techniques when necessary.
- We develop a scheme for organising and specifying use contexts/scenarios and related roles for components in their interfaces. It allows the specification of multiple interface configurations, with each for a use context/scenario. For one interface configuration, multiple ports can be defined to capture the roles that the component plays in the given context.
- We introduce a constraint-based approach to the specification of interaction protocols for software components. Instead of always specifying the full interaction semantics of a component, the approach allows incremental specification of interaction semantics in the form of constraints, following the philosophy of “light-weight” semantics advocated by Perry [19, 20]. The constraints specification concerns only the interface elements of the component, and is expressed in terms of their temporal properties or relationships. The approach distinguishes general, configuration-specific and role-specific constraints.

Within the proposed framework for rich interface specification, the behavioural semantics of interface elements is specified following proven approaches, such as those in Eiffel [17] and Catalysis [6], based on UML’s Object Constraint Language (OCL) [24]. The framework reserves space for the specification of quality properties. But the specific schemes used to specify these properties is not considered in this article as each quality aspect requires its own scheme of characterisation or specification and only limited work has been done in this area.

The article is organised as follows. In Section 2, we present a system architecture for telecommunications software, which provides the examples used in subsequent sections. In Section 3, we introduce the framework for rich component interface specification, and discuss its various aspects in detail. In Section 4, we highlight a number of applications for the rich interface specification framework. In Section 5, we review related work before providing some concluding remarks in Section 6.

## 2 An architecture for telecommunications software

In this section, we present an example architecture for telecommunications software systems. We will use it to illustrate our framework for rich component interface specification in the next section.

Software is an essential part of telecommunications systems. It implements the required telecommunications protocols and performs various functions like transmission, switching and access management. Common to many of the telecommunications software systems is a particular system architecture, involving different kinds of system modules. One system module in the system architecture is the *central manager* (CM) that monitors and coordinates all other modules in the system. Another class of modules are *management and service* (MS) units, each type of which is responsible for a specific aspect of the system’s functionality. There may be different types of MSs in a system. The third class of modules are *service arbitrators* (SAs) that not only have functionality of their own like MSs, but also coordinate the services provided by a certain group of service modules. The different types of *service modules* (SMs) provide specific services and are coordinated by service arbitrators.

All the system modules contain *meta-data* describing themselves. This is to facilitate system evolution, upgrade and configuration dynamically at run-time. When a module first comes into service, its meta-data is communicated to the CM module so that the CM is aware of the new module’s existence and is able to manage its activities. An MS module is relatively independent

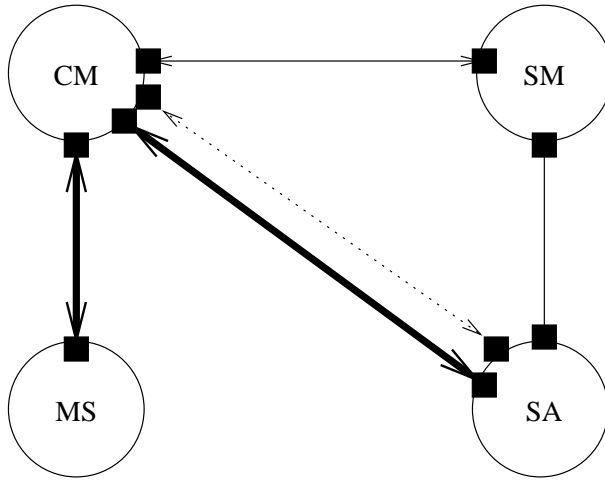


Figure 1: A Simplified Telecommunications System Architecture

and can communicate to the CM about its meta-data and interact with other system modules (including the CM) regarding its functionality. However, some of the meta-data of a service module (SM) resides in its corresponding service arbitrator, due to the limitation in space and processing capability of the hardware device that the service module resides. In fact, this is a major reason why we distinguish service arbitrators from service modules and MS modules. For the same reason and other coordinating purposes, a service arbitrator coordinates and presents to the rest of the system, some of the services provided by the service modules under its control.

The behaviour of a telecommunications software system and its components (CM, MSs, SAs and SMs) depends on the system's use context. The system (as a higher-level component) may be used as a node in a ring configuration or a point-to-point configuration of a telecommunications network. In these two contexts, the system (node) and its components should behave accordingly and differently.

Figure 1 shows the architecture of a specific telecommunications software system, much simplified for the purpose of discussion in this article. It contains a CM component, an MS component, an SA component and an SM component, with reduced interactions. The thick firm arrows indicate normal interactions between components. The thin firm arrow indicates reduced interactions between CM and SM. The dotted arrows indicate SA's interactions with CM on behalf of SM. The normal line indicates SA's access to SM. The filled squares in the figure indicate the logical interaction points of the components. Note that the diagrammatic elements in Figure 1 do not suggest a specific notation for architecture representation, which is outside the scope of this article.

### 3 Component interface specification

Interface specification provides a basis for the development, management and use of software components. As shown in Figure 2, our framework for component interface specification addresses the following aspects: signature (syntax), organisation/configuration (structure), behaviour (semantics), interaction (protocols/constraints), and quality.

At the bottom level of the interface specification, there is the *signature* of the component, which forms the basis for the component's interaction with the outside world and includes all the necessary mechanisms or elements for such interaction (i.e., attributes, operations and events). The next level up is about the structural organisation of the interface in terms of the component's roles in given contexts of use, i.e., *configurations*. As such, the component interface may have different *configurations* depending on the use contexts, each configuration may consist of a number of ports

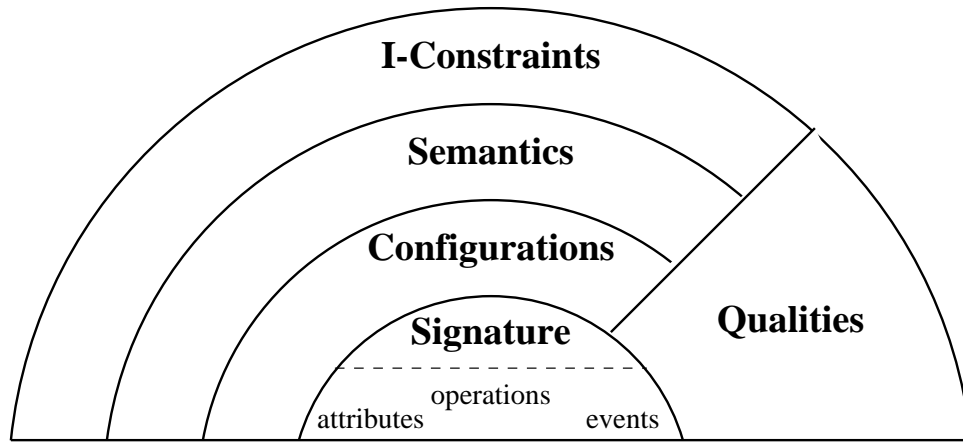


Figure 2: Structure of Component Interface Specification

reflecting the roles that the component plays relative to neighbouring components, and a port uses a number of interface elements that belong to the signature.

The third level of the interface specification concerns the *semantics* of the individual signature elements, capturing their precise behaviour. At the fourth level are the *interaction constraints* defining the interaction protocols of the component. Observing these constraints is necessary to ensure the proper use of the component in a given context.

The fifth aspect of the interface specification is about the characterisation of the component's *non-functional* or *quality properties*, such as those regarding performance, reliability and security. The non-functional properties occupy a special place in this component interface structure, and may have dependency relationships with other aspects of the interface. In general, each of the quality attributes requires its own model of characterisation and corresponding specification scheme in the interface.

The interface signature and behavioural semantics define the overall functional capability of the component, which should be conformed to by any use of the component. The interface configurations and interaction constraints concern the proper use of the component in perceived application contexts. The quality properties provide the basis for assessing the suitability or usability of the component (relative to given application contexts).

In the following subsections we discuss the five aspects of component interface specification in detail.

### 3.1 Interface signature

Fundamental to a component's interface is its signature that forms the basis of all other aspects of the component interface. The component interface signature lists all the necessary elements that are used for the component's interaction with the outside world. As commonly recognised in existing industrial standards and practice, including CORBA, JavaBeans and COM, the interface signature of a component comprises *attributes*, *operations* and *events*. Our scheme for interface signature specification is, in essence, a consolidation of the well accepted standards and practice with consideration of both provided and required services.

A software component may have a number of attributes *externally observable*. These attributes form an essential part of the component interface, i.e., the *observable structural elements* of the component. The users, including people and other software components, may use (i.e., observe and even change) their values, to understand and influence the component's behaviour. A common use of component attributes is for component customisation and configuration at the time of connection [23]. Component attributes can also be used *during* the component's interaction in the sense of

attributes in CORBA IDL. It should be noted that certain component attributes can only be observed but not changed, i.e., read-only attributes.

Another aspect of a component signature is the operations, which are the main mechanism for the component to interact with the outside world. The operations capture the dynamic behavioural capability of the component, and represent the services/functionality that the component provides.

Besides proactive control (usually in the form of explicit operation invocation or message passing), another form of control used to realise system behaviour is reactive control (usually in the form of event-driven implicit operation invocation or message passing). It is often the case that certain aspects of a system are better captured through proactive control via operations, while other aspects of the system are better captured in the form of reactive control via events. To facilitate reactive control, a component may generate events from time to time, which other components in the system may choose to respond to. In this type of event-based component interactions, there may be none or many responses to an event, and they may change as time goes on. As such, this model of interaction allows communication channels to be established dynamically [11], and is particularly suited to dynamic system (re-)configuration.

The above discussion is from the viewpoint of the interface elements *provided* by the component. The component interface specification should also include the elements that it requires from other components. The component can retrieve and even change the required attributes, invoke the required operations, and observe and possibly respond to the required events. In general, the specification of the required interface elements take a form similar to that of the provided ones.

For illustration purpose and to provide a basis for the discussion in the rest of the paper, we consider the interface signatures of the components in the telecommunications software example. The central manager (CM) component provides a range of services to other system components<sup>1</sup>. When a system module comes into service, the CM component first requests that module's signature, then initialises the module by setting its alarm, performance and network configuration attributes (among others). After that, the CM will enable the module for normal service, and then the module will be able to report to the CM about its alarm and performance status (among other information). Note that the performance-related attribute initialisation and status reporting of service modules are done through their proxies (namely, service arbitrators). In addition, the CM component can raise a system testing event and all other modules are supposed to respond by lighting their testing LEDs. CM's interface signature for the above activities looks as follows:

```
COMPONENT cm {
  SIGNATURE {
    PROVIDES {
      report_signature(IN Module m_id, IN Sig sig);
      report_alarm_data(IN Module m_id, IN AlarmData alarmData);
      report_perf_data(IN Module m_id, IN PerfData perfData);

      EVENT e_system_testing; /* all modules respond by lighting LEDs */
    }
    REQUIRES {
      ATTRIBUTE BOOLEAN enabled;

      request_signature();
      set_alarm_attributes(IN Alarm alarm);
      set_perf_attributes(IN Perf perf);
      set_proxy_perf_attributes(IN SM sm_id, IN Perf perf);
      set_ring_attributes(IN Ring_protect protect);
    }
  }
}
```

---

<sup>1</sup>Here we only show part of the services for simplicity purpose, and similarly for other components.

```

        set_pTp_attributes(IN PtP_protect protect);
        system_testing(); /* to respond to e_system_testing */
    }
}
}

```

Note that the CM component provides three operations and one event. In general, an event may carry with it a number of value parameters for use by the respondents, but the `e_system_testing` event here does not have such parameters. The CM component also requires two attributes and seven operations from the managed components regarding their status, initialisation and testing. Note that it is not necessary for every managed component to provide all the required elements. For example, the operations for setting network configuration attributes, `set_ring_attributes` and `set_pTp_attributes`, are only relevant to the SA module, but they need to be listed in the section for required services.

The MS module will respond to the signature request, attribute initialisation and testing event from the CM module. It also provides an attribute `enabled` for CM to set after initialisation.

```

COMPONENT ms {
    SIGNATURE {
        PROVIDES {
            ATTRIBUTE BOOLEAN enabled;

            request_signature();
            set_alarm_attributes(IN Alarm alarm);
            set_perf_attributes(IN Perf perf);
            system_testing(); /* for system testing */
        }
        REQUIRES {
            report_signature(IN Module m_id, IN Sig sig);
            report_alarm_data(IN Module m_id, IN AlarmData alarmData);
            report_perf_data(IN Module m_id, IN PerfData perfData);

            EVENT e_system_testing; /* event to be responded for testing */
        }
    }
}
}

```

For communication with CM, the MS component provides an attribute and four operations to CM, and requires three operations and an event from CM.

The SA module has functionality similar to MS, but without the performance attributes and data. In addition, it manages the SM module's performance attributes and data in a proxy role. Furthermore, the SA module will behave differently regarding its protection scheme, depending on whether the system is in a ring or point-to-point network configuration. The CM module communicates to the SA the configuration information and sets its protection attributes.

```

COMPONENT sa {
    SIGNATURE {
        PROVIDES {
            ATTRIBUTE BOOLEAN enabled;

            request_signature();
            set_alarm_attributes(IN Alarm alarm);

```

```

    set_proxy_perf_attributes(IN SM sm_id, IN Perf perf);
    set_ring_attributes(IN Ring_protect protect);
    set_pTp_attributes(IN PtP_protect protect);
    system_testing(); /* for system testing */
}
REQUIRES {
    ATTRIBUTE Perf performance;
    readonly ATTRIBUTE PerfData perfData;

    report_signature(IN Module m_id, IN Sig sig);
    report_alarm_data(IN Module m_id, IN AlarmData alarmData);
    report_perf_data(IN Module m_id, IN PerfData perfData);

    EVENT e_system_testing; /* event to be responded for testing */
}
}
}

```

For communication with CM, the SA component provides one attribute and six operations, and requires three operations and an event. SA also requires two attributes from the SM component for coordinating its performance related information (see below for details).

The SM module interacts with CM directly about its signature and alarm information, and responds to CM's system testing event. However, its performance initialisation and data reporting are coordinated by SA, and are represented by a `performance` attribute for initialisation and a read-only `perfData` attribute for data reporting in SM.

```

COMPONENT sm {
    SIGNATURE {
        PROVIDES {
            ATTRIBUTE BOOLEAN enabled;
            ATTRIBUTE Perf performance;
            readonly ATTRIBUTE PerfData perfData;

            request_signature();
            set_alarm_attributes(IN Alarm alarm);
            system_testing();
        }
        REQUIRES {
            report_signature(IN Module m_id, IN Sig sig);
            report_alarm_data(IN Module m_id, IN AlarmData alarmData);

            EVENT e_system_testing; /* event to be responded for testing */
        }
    }
}
}

```

For communication with CM, the SM component provides the attribute `enabled` and three operations, and requires two operations and one event. Note that the `set_perf_attributes` and `report_perf_data` operations are missing from the interface. In fact, they are represented by the two attributes: `performance` and `perfData`, which are set and retrieved respectively by the SA component.

## 3.2 Interface configurations

The interface signature of a component lists all the provided and required elements relevant to its interaction with other components. To provide better support for the understanding and use of the component, some usage-oriented structural characteristics need to be specified in its interface. It involves two aspects: (1) the component plays different roles in a given context or scenario of use, and (2) the component may be used in different types of context.

In a particular use scenario, a component (called the *focal component*) usually interacts with a number of other components (called the *neighbouring components*), and plays specific roles relative to them. The interactions between the focal component and its neighbouring components may differ depending on the neighbour and its related perspectives. When interacting with a particular neighbouring component from a specific perspective, for example, the focal component may only make visible or use, certain attributes, operations and events. This also implies that the neighbouring component does not need to provide all the required services or use all the provided services, of the focal component. On the other hand, one interface element may be used to interact with more than one of the neighbouring components. To make this organisational information available to the user, we need to define perspective/role-oriented interaction ports for the focal component, i.e., *an interface configuration*. Each *port* in an interface configuration contains those and only those (provided and required) elements that are relevant to the interaction between the two components concerned from the given perspective.

The use scenarios provide the contexts of use for the component. A component may be used in different scenarios and has different role partitions in these scenarios. For a component, therefore, there may be the need for different interface configurations, with each configuration for a type of scenarios in which the component is to be used. In principle, an interface configuration should be defined in terms of both the component and the use scenario.

When a component is designed, the designer usually has one or more use scenarios in mind. Therefore, a few configurations may be defined for the component interface. Later on when a new type of use scenario is discovered, a new configuration may be added to the component interface.

The importance of interface configuration can not be over emphasised. It serves to relate a component to its contexts of use. In fact, many of the requirements for a component are derived from its use scenarios. The roles that a component plays in a use scenario are vital to the architectural design of the enclosing system. They provide the basis for defining the interactions between the components of the system and realising the system functionality. Interface configuration enables the relative independent development of the system components with clearly defined interfaces as well as requirements.

**Interface configuration of CM.** In the reference example, the CM component plays four roles in a ring network setting: **MS management** relative to MS, **SA management** relative to SA, **SM management proxy** relative to SA, and **SM management** relative to SM. These roles form the basis of an interface configuration for CM. Note that CM plays two roles relative to SA, i.e., the **SA management** and **SM management proxy** roles. Although being between the same pair of components, these two roles each have their own specific functionality.

CM's **MS management** role defines the interface for interaction between CM and MS from CM's perspective. All the interface elements in the corresponding interface port are drawn from CM's interface signature, with the particular understanding that the required elements in the port are to be provided by MS.

```
PORT ms_management {
    /* PROVIDES */
    report_signature(IN MS m_id, IN Sig sig);
    report_alarm_data(IN MS m_id, IN AlarmData alarmData);
}
```

```

    report_perf_data(IN MS m_id, IN PerfData perfData);
    e_system_testing;
    /* REQUIRES */
    BOOLEAN enabled;
    request_signature();
    set_alarm_attributes(IN Alarm alarm);
    set_perf_attributes(IN Perf perf);
    system_testing();
}

```

Note that the parameter type `Module` of the operations is further specialised into `MS` in this role, and that not all the required elements are relevant to this role. In general, the category, qualification, type and parameter information of an interface element does not need to be repeated in a configuration if there is no change or specialisation; but it can be provided in part or in whole for clarity.

The `SA` management role of `CM` in a ring network setting looks as follows:

```

PORT sa_management {
    /* PROVIDES */
    report_signature(IN SA m_id, IN Sig sig);
    report_alarm_data(IN SA m_id, IN AlarmData alarmData);
    e_system_testing;
    /* REQUIRES */
    BOOLEAN enabled;
    request_signature();
    set_alarm_attributes(IN Alarm alarm);
    set_ring_attributes(IN Ring_protect protect);
    system_testing();
}

```

On this role, there are no performance related operations due to the nature of `SA`. The operation for setting the point-to-point network configuration, `set_pTp_attributes`, is excluded from the required services as it is not needed.

The `SM` management proxy role of `CM` is only about `SM`'s performance information.

```

PORT sm_management_proxy {
    /* PROVIDE */
    report_perf_Data(IN SM sm_id, IN PerfData perfData);
    /* REQUIRE */
    set_proxy_perf_attributes(IN Perf perf);
}

```

Note that although this role is relative to `SA`, it is about the `SM` coordinated by `SA`.

The `SM` management role of `CM` is as follows:

```

PORT sm_management {
    /* PROVIDES */
    report_signature(IN SM m_id, IN Sig sig);
    report_alarm_data(IN SM m_id, IN AlarmData alarmData);
    e_system_testing;
    /* REQUIRES */
    BOOLEAN enabled;
    request_signature();
}

```

```

        set_alarm_attributes(IN Alarm alarm);
        system_testing();
    }

```

Note that the `performance` and `perfData` attributes of SM are not relevant to this role or the CM interface signature. In fact, they are related to the interactions between SA and SM.

With the above four roles, the CM component with a configuration for the ring network setting will have the following interface specification:

```

COMPONENT cm {
    SIGNATURE { ... }
    CONFIGURATION ring {
        PORT ms_management { ... }
        PORT sa_management { ... }
        PORT sm_management_proxy { ... }
        PORT sm_management { ... }
    }
}

```

As outlined earlier, the CM module may manage many modules of the MS, SA and SM types in a full-scale system. This suggests that in CM's `ring` configuration of such a system there may be many, dynamically determined instances for each of the roles or ports defined above. A straightforward extension can be made to accommodate the specification of interface configurations with multiple instances of same roles/ports. For simplicity, we focus on configurations with single instance roles in this article.

**Interface configuration of MS.** The MS component plays a single role relative to CM: `MS management`. The definition of this role is similar to that of the `MS management` in CM except that the `PROVIDES` and `REQUIRES` sections are swapped.

**Interface configuration of SA.** The SA component plays three roles: `SA management` and `SM management proxy` relative to CM, and `SM operational management` relative to SM. The `SA management` and `SM management proxy` roles of SA mirror their counter parts in CM. In the `SM operational management` role, the `performance` and `perfData` attributes (of SM) are made available to SA for direct manipulation.

```

COMPONENT sa {
    SIGNATURE { ... }
    CONFIGURATION ring {
        PORT sa_management { ... }
        PORT sm_management_proxy { ... }
        PORT sm_operational_management {
            /* REQUIRES */
            Perf performance;
            PerfData perfData;
        }
    }
}

```

**Interface configuration of SM.** The SM component has two roles: `SM management` relative to CM and `SM operational management` relative to SA. These two roles mirror their counter parts in CM and SA respectively.

**Another set of interface configurations.** The above component interface configurations are designed for use in the situation where the system is used as a node in a ring network. When the system is used as a node in a point-to-point network, the roles of the components have a much reduced capacity while the types and number of roles remain the same<sup>2</sup>, and a different protection scheme is required. In terms of interface definition, the main differences are reflected in the SA management roles of CM and SA. In CM, the SA management role would look like:

```

PORT sa_management {
    /* PROVIDES */
    report_signature(IN SA m_id, IN Sig sig);
    report_alarm_data(IN SA m_id, IN AlarmData alarmData);
    e_system_testing;
    /* REQUIRES */
    BOOLEAN enabled;
    request_signature();
    set_alarm_attributes(IN Alarm alarm);
    set_pTp_attributes(IN PtP_protect protect);
    system_testing();
}

```

Note the changes related to the point-to-point setting, i.e., the `set_ring_attributes` has been replaced by `set_pTp_attributes`.

In general, the MS and SM modules can use the same interface configuration for both ring and point-to-point networks, while CM and SA have separate configurations for these situations. For example, the CM interface definition will have the following structure with two configurations:

```

COMPONENT cm {
    SIGNATURE { ... }
    CONFIGURATION ring { ... }
    CONFIGURATION pTp { ... }
}

```

The interface configurations of components are a higher-level abstraction than the interface elements. They provide an important link to the supposed contexts of use for the components, and therefore greatly aid the understanding and use of the components.

### 3.3 Behaviour of interface elements

The interface signature spells out the individual elements of interaction in mostly syntactic terms. The interface configurations introduce usage-oriented structural or organisational characteristics. To understand fully the component's behaviour, additional semantic information about the interface elements is required, and should be specified explicitly. For an attribute, its value range may be further restricted on the basis of its type. The value or value change of an attribute may be constrained in relation to the component state.

An event may have specific characteristics, including whether it is compulsory or optional and whether it is singular or plural. A compulsory event must have at least one respondent in the system. An optional event may have zero or more respondents. A singular event may have at most one respondent, i.e., consumed by the respondent. A plural event is not consumed by the respondents and therefore can have many respondents. Furthermore, there may be further value range restrictions on an event's parameters.

---

<sup>2</sup>In general, the types and number of roles of a component may change depending on the contexts of use.

The use of pre- and post-conditions for defining the behaviour or semantics of operations has been well studied, such as those used in Eiffel [17] and Catalysis [6]. Here we follow such an approach and use UML's Object Constraint Language (OCL) to specify the pre- and post-conditions of operations [24].

The behaviour of a provided service (interface element) is specified relative to the component concerned, while the behaviour of a required service (interface element) is specified relative to the component providing the service. As an example, we show below the behavioural specifications of CM's interface elements:

```

COMPONENT cm {
  SIGNATURE {
    PROVIDES {
      report_signature(IN Module m_id, IN Sig sig) {
        PRE: true;
        POST: self.retrieve_signature(m_id) = sig};
    };
      report_alarm_data(IN Module m_id, IN AlarmData alarmData) {
        PRE: self.enabled(m_id);
        POST: self.retrieve_alarm_data(m_id) = alarmData;
    };
      report_perf_data(IN Module m_id, IN PerfData perfData) {
        PRE: self.enabled(m_id);
        POST: self.retrieve_perf_data(m_id) = perfData;
    };

      EVENT e_system_testing {optional, plural};
        /* all modules respond by lighting LEDs */
    }
  REQUIRES {
    ATTRIBUTE BOOLEAN enabled {
      enabled@^ = false;
    };

    request_signature() {
      PRE: true;
      POST: self.sig_request() = received;
    };
    set_alarm_attributes(IN Alarm alarm) {
      PRE: not self.enabled;
      POST: self.retrieve_alarm_attributes() = alarm);
    };
    set_perf_attributes(IN Perf perf) {
      PRE: not self.enabled;
      POST: self.retrieve_perf_attributes() = perf);
    };
    set_proxy_perf_attributes(IN SM sm_id, IN Perf perf) (
      PRE: not self.sm_enabled (sm_id);
      POST: self.retrieve_perf_attributes(sm_id) = perf);
    };
    set_ring_attributes(IN Ring_protect protect) {
      PRE: not self.enabled;

```



the operation must not be `true`.

In concurrent and distributed applications, assuming the services provided by a component are atomic and are executed as transactions is not always practical or true [3]. Furthermore, a given application context may dictate a restricted way in which a component is to be used. It is beneficial that these restrictions on the component’s capability in the given use context are made clear to its users. In general, the dependency or synchronisation of a component’s interface elements concerns the use or usage of the component, and defines the *interaction protocols* of the component. Making these protocols explicit helps the user to understand the component and facilitates their checking and enforcement. For example, two attributes may be inter-related in terms of their value settings. One operation may have to be immediately invoked after another operation’s completion.

In specifying a component’s interaction protocols, we adopt a constraint-based approach. The basic assumption of the approach is that when there are no interaction constraints specified for a component, any interaction sequences and states are allowed subject to the component’s signature and configuration. As such, certain interactions with the component are successful, while others generate exceptions, errors or unpredictable behaviours. The reason is that this latter class of interactions violate the semantic specification of the relevant interface elements and their interaction protocols. Therefore, the set of interaction scenarios allowed is much larger than the actual valid set of interaction scenarios. When an interaction constraint is specified and checked (before actual interaction), it will eliminate certain invalid interaction scenarios that are allowed by the basic assumption. In essence, it reduces the allowable set towards the valid set of interactions. When more interaction constraints are added, this approximation becomes more accurate. Theoretically, when all the possible interaction constraints are specified, the allowable set will be the valid set.

The constraint-based approach for specifying interaction protocols allows incremental and flexible specification, checking and enforcement of interaction constraints. That is, there is no requirement that all interaction constraints must be specified. In a sense, it advocates the line of “specifying as much as you require or see fit”, and follows the spirit of “light-weight” semantics in Inscape [19, 20]. Therefore it is more practical to use than always requiring a full specification.

**Constructs for constraint specification.** In specifying the interaction constraints, we introduce two constructs to express the relationships between interface elements. The first has the following form:

*attribute@time*

which gives the value of the *attribute* at the given *time*. The *time* can be  $\wedge$  for upon the connection of component or role, an event for its firing, `PRE(op)` for before the invocation of the operation *op*, or `POST(op)` for after the completion of the operation *op*. For example, `enabled@ $\wedge$  = false` states that the attribute `enabled` has a `false` value upon connection. `enabled@PRE(report_alarm_data)` means that `enabled` being `true` is a precondition of the operation `report_alarm_data`.

The second construct has the following form

*action tr action*

where *action* can be  $\wedge$  for component or role connection, an operation, an event, or `attribute.SET(value)` or `attribute.GET` for setting and retrieving the `attribute`’s value; *tr* is a temporal relational operator: (immediately) `proceeds` or `before`. The construct states that the two actions satisfy the given temporal relationship. For example, ( $\wedge$  `proceeds request_signature`) means that the first operation after connection is `request_signature`. The constraint (`set_alarm_attributes before enabled.SET(true)`) means that the `set_alarm_attributes` operation is carried out *before* the `enabled` attribute is set to `true`, there may or may not be other actions between them, and in fact, there is no guarantee that the second action will ever happen. Note that concurrency (||) between actions is assumed by default, but we may state that two actions can not be carried out concurrently.

**Types and placement of constraints.** An interaction constraint may be about only provided elements, only required elements, or both provided and required elements. Certain constraints may be general to all interface configurations, while others are specific to individual configurations or roles. For example, the CM component in the telecommunications example is connected to three other components, each of these other components provides its own `enabled` attribute and `set_alarm_attributes` operation. A constraint may state that a component's `set_alarm_attributes` operation must be carried out before its own `enabled` attribute is set to `true`. Furthermore, CM's `report_alarm_data` operation can be invoked by a component (MS, SA or SM) only after that component's `enabled` attribute is set to `true`.

Interaction constraints of a component should be specified in the relevant sections of its interface specification to facilitate easy understanding. That is, constraints that are independent of configurations and specific to the provided services are specified in the `PROVIDES` section of the signature; constraints that are independent of configurations and specific to the required services are specified in the `REQUIRES` section; constraints that are independent of configurations and concern both provided and required services are specified in a new `CONSTRAINTS` section in the signature; role-specific constraints are specified in the relevant role; configuration-specific and cross-role constraints are specified in a new `CONSTRAINTS` section in the relevant configuration.

**Configuration-neutral constraints.** As an example, we consider two general constraints on the provided services of the MS component. The first states that the attributes initialisation operations `set_alarm_attributes` and `set_perf_attributes` must be performed before the module is enabled for normal service. The second states that the system testing operation may only be performed after the module is enabled for service.

```
COMPONENT ms {
  SIGNATURE {
    PROVIDES {
      ATTRIBUTE BOOLEAN enabled;

      request_signature();
      set_alarm_attributes(IN Alarm alarm);
      set_perf_attributes(IN Perf perf);
      system_testing();

      CONSTRAINT
        (set_alarm_attributes, set_perf_attributes) before enabled.SET(true);
      CONSTRAINT enabled@PRE(system_testing);
    }
  }
}
```

In fact, the above constraints should also be reflected in the relevant operations preconditions if they are specified (see also the behavioural specifications of CM's required services). Also note that SA and SM have similar constraints.

In general, there will be very few general constraints that are specific to the required services or are about both required and provided services, because the required services may be provided by different components and the distinctions are only made clear in the configurations.

**Role-specific constraints.** In CM's `MS management` role, we introduce three constraints. The first states that the first operation on the role is `request_signature` and is immediately followed by the `report_signature` operation. The second states that on this role, the alarm and performance

initialisation operations are carried out before MS is enabled. The third constraint states that the two reporting operations can only be invoked when MS is enabled.

```

PORT ms_management {
    /* PROVIDES */
    report_signature(IN MS m_id, IN Sig sig);
    report_alarm_data(IN MS m_id, IN AlarmData alarmData);
    report_perf_data(IN MS m_id, IN PerfData perfData);
    e_system_testing;
    /* REQUIRES */
    BOOLEAN enabled;
    request_signature();
    set_alarm_attributes(IN Alarm alarm);
    set_perf_attributes(IN Perf perf);
    system_testing();
    /* CONSTRAINTS */
    CONSTRAINT ^ proceeds request_signature proceeds report_signature;
    CONSTRAINT (set_alarm_attributes, set_perf_attributes) before
        enabled.SET(true);
    CONSTRAINT enabled@PRE(report_alarm_data) and
        enabled@PRE(report_perf_data);
}

```

Note that these constraints are specific to the `MS management` role of CM. The second constraint (in this CM's role) mirrors a constraint on MS's provided services (see above). The third constraint synchronises the value of `enabled` in MS (required) and the invocation of the data reporting operations on CM. This corresponds to the preconditions on the relevant operations, where we assumed that MS's status can be retrieved from CM using an auxiliary function (see Section 3.3). It is obvious that for this purpose, the constraint specification is straightforward and uses only interface elements.

CM's `SA management` role and `SM management` role have constraints similar to those of its `MS management` role, except that they are only about elements related to alarm.

**Configuration-specific constraints.** From CM's perspective, the coordination about SM's performance attributes and data crosses both the `SM management` role and the `SM management proxy` role. The relevant coordination constraints are therefore constraints of the configuration rather than those of individual roles.

```

COMPONENT CM {
    SIGNATURE { ... }
    CONFIGURATION ring {
        PORT ms_management { ... }
        PORT sa_management { ... }
        PORT sm_management_proxy { ... }
        PORT sm_management { ... }
        CONSTRAINTS /* configuration constraints */
            (sm_management_proxy.set_proxy_perf_attributes) before
                sm_management.enabled.SET(true);
            (sm_management.enabled)@PRE(sm_management_proxy.report_perf_data);
    }
}

```

The first constraint indicates that the `set_proxy_perf_attributes` operation in the `SM management proxy` role must be carried out before the SM module is enabled. The second constraint states that the `report_perf_data` operation in the `SM management proxy` role can only be carried out after the SM module is enabled.

### 3.5 Quality properties

Another aspect of a component is its non-functional or quality properties. In the context of building systems from existing components, the characterisation of the components' quality properties and their impact on their enclosing systems are particularly important because the components are usually provided as blackboxes. However, there is not much work done in this area. Therefore, there is an urgent need to develop characterisation models for each of the quality attributes like security, performance and reliability, in the context of software components and composition.

For a particular quality attribute, two issues need to be addressed: (1) how to characterise the relevant quality properties for a component, and (2) how to analyse the properties' impact on the enclosing system in a given context of use (i.e., in the context of a system architecture). A related issue is whether the characterisation of the quality properties will change in different contexts of use. Currently, we are investigating the security aspect of software components and its impact on system composition in the context of developing distributed electronic commerce systems [10]. Another example of quality attribute characterisation is about performance [22]. The quality property specification will be dependent on the specific characterisation models developed. While we do not have definite models available yet, the interface definition framework proposed in this article can be extended to accommodate new models concerning quality properties.

### 3.6 Discussion

In this subsection, we further clarify a few issues regarding the model for rich interface specification of software components presented above.

**Richness of interface specification.** The interface specification model involves five aspects: signature, configurations, behavioural semantics, interaction constraints, and quality properties. It should be emphasised that for a given component, its interface specification does not have to include all these aspects, and furthermore it is not necessary to have a complete specification of some included aspect. In fact, it is unrealistic and impractical to expect that all components have complete specification for every aspect of their interface, even in the presence of a standard. As expected, the most basic aspect of a component's interface specification is its interface signature. The existence and completeness of other aspects in its interface specification are optional.

In a given context, a component can be used as long as its available interface information meets the requirements of the context. If a use scenario does not demand the behavioural specification and security characterisation of a component, for example, a component without these specifications can be used as long as the other interface information meets the requirements of the scenario. With components having varied levels of interface specification, it is at the time of use that we need to assess the adequacy of a component's interface information relative to the use context. This flexibility enables practical and non-intrusive introduction of the advanced aspects of component interface specification.

**Interaction constraints versus behavioural semantics.** We have seen that there is certain overlap in scope between interaction constraints/protocols and behavioural semantics of interface elements, especially regarding attribute values appearing in operations' pre-conditions. However, interaction constraints do offer additional information regarding the dependency and synchronisation between interface elements, which delivers practical benefits. First, this information helps the

proper understanding and use of components. With this information, the user of a component can easily avoid the interaction scenarios that lead to exceptions, errors or unpredictable behaviours. The user can also easily appreciate the additional interaction constraints specific to the context of use.

Second, the explicit specification of the interaction constraints facilitates static and dynamic checking and enforcement of interaction dependency and synchronisation. At the time of deciding to use a component, its suitability regarding inter-component interaction can be checked based on its interaction constraints and the interaction requirements of the use context. At run time, the interactions with a component can be monitored and its interaction constraints be enforced by the execution environment to achieve expected system behaviour.

Ideally, the behavioural semantics of a component's interface elements should be available and be observed in its use. However, its specification is more involved. The checking of semantic compatibility between a component and its use context requires the service of an interactive theorem prover. In reality, we expect that the interaction constraints will be used more often than the behavioural semantics, and a component interface specification is more likely to include signature, configurations and interaction constraints than behavioural semantics. The interaction constraints provide a middle ground between with and without semantic specification for component interface, and allows the easy specification, checking and enforcement of some "light-weight" semantics [20, 4].

**Component specification versus architecture design.** This paper is about interface specification of components, rather than design of system architecture involving components. To help the exposition of the interface specification model, however, the examples used in the discussion of interface specification are all drawn from a reference system architecture to give the reader a single context of understanding. Other simplifications deployed include the name correspondence for required and provided elements of interacting components, for configurations across components and for roles of corresponding configurations in different components. From the interface specification viewpoint, such economy is in fact not necessary because all the components are regarded as *independent* software units that are not necessarily related. The corresponding elements, configurations and roles do not have to have same names. The match of names between components and the linking between events and operations are carried out at the system assembly time according to a system architecture, which is outside the scope of this paper. Interested readers are referred to architecture-directed assembly techniques such as those used in [13] and [25].

Our focus on interface specification of independent components does affect the structure and information of the interface specification. In particular, one may have already sensed a certain amount of interface information duplication between corresponding components. For example, a required element in one component corresponds to a provided element in another component; a configuration or role of one component has its mirror in another component; even the interaction constraints are duplicated in the relevant parts of corresponding components. In general, this is necessary because the components (and therefore their interface specifications) are independent of each other. If all the components are specified and designed in the context of a single system architecture, on the other hand, one may argue that certain interaction information, especially the configurations and interaction constraints can be specified separately from the components (as part of the architecture) to reduce the duplication. However, when we consider a component alone (e.g., by its own design team), all the relevant interaction information in the architecture still needs to be considered or mapped to the component's scope. In fact, many architecture description approaches also adopt the component-focused interaction specifications, such as Rapide [13] and Wright [1].

**Concepts and mechanisms for component interface specification.** In organising interface structure, we have introduced context-oriented configurations and role-based ports. These are higher-level concepts than interface elements, and make component understanding easier. In

particular, they help the architecture-directed system assembly, where configurations, roles/ports as well as elements are used for inter-component connection.

In specifying the interaction relationships, we have introduced two temporal operators: **proceeds** and **before**. A stronger version of **before** is **leadsTo**: (**a leadsTo b**) requires that action **b** eventually happen whereas (**a before b**) does not. In addition, further variations or extensions to the temporal relational operators are possible. For example, both **before** and **leadsTo** may have versions requiring that the action instances are pair-wise disjoint.

In constraining the number of times that an action can be performed, we can use the following constructs in behavioural and interaction constraint specifications: (**not action**) and (**once action**), where the *action* may have special valued parameters to have fine-grained control. For example, the **enabled** attribute in the telecommunications example can be further restricted as follows:

```
ATTRIBUTE BOOLEAN enabled {
  enabled@^ = false;
  not enabled.SET(false);
  once enabled.SET(true);
};
```

The second constraint states that the **enabled** attribute can not be *set* to **false** (although initially it has a **false** value). The third constraint states that the **enabled** attribute can only be set to **true** *once*.

We have not introduced mechanisms that help to structure the interface *specification*. For example, we could introduce an inheritance mechanism for defining new configurations or ports based on existing ones. Not including such mechanisms is a deliberate choice because we believe that interface specifications should be kept as simple and straightforward as possible so not to obscure their understanding.

## 4 Applications

While having a general purpose of aiding component understanding and use, rich interface specification for software components has a variety of specific applications. In this section, we highlight a few of such applications.

**Component-based system assembly.** When we use existing software components to assemble a system, we need to understand what these components do, whether they meet the requirements of the system, and how they fit into the system architecture. The rich interface specification provides the basis for assessing a component's suitability in the given system context. Based on its interface specification, we can reason about whether a component meets the syntactic, structural, semantic, interaction and quality requirements for an envisaged component in the system architecture. Any discrepancies or mismatches can be identified, analysed and possibly eliminated through adjusting the system architecture or introducing necessary adaptors [25]. In particular, we can deduce or predict the quality properties of entire system based on those of the components used and those of the system architecture, especially when the *quality requirements* for the envisaged components of a system architecture are not clear or available. At the time of system assembly, the relevant configurations of the components used are selected, the ports of the configurations are connected, and the attributes, operations and events of corresponding ports are mapped, in the context of the system architecture [21].

**Architecture-directed system design.** The rich interface specification approach for components is equally applicable to architecture-directed system design that does not have existing components available for use, sometimes called *forward engineering*. In such a situation, the system

architecture is defined in terms of envisaged components, and the requirements for these components are codified in the form of their rich interface specifications. The exposition of the telecommunications example in this article illustrates the relevant part of this process.

Many features of the interface specification approach reported in this article was, in fact, initially developed and applied in the context of a telecommunications software project at Fujitsu Australia. Combined with object oriented analysis techniques such as scenario analysis, it delivered immediate benefits: (1) clear definition of the subsystems/components' capabilities through their interfaces; (2) clear identification of the interactions between the subsystems; (3) early analysis of system behaviour at architectural level. This has significantly reduced the interaction between the teams responsible for the various subsystems, and avoided many of the architectural changes later in the development cycle that would be normally experienced in traditional projects. This experience has shown that the proposed rich interface specification framework has provided methodological support for architecture-directed and component-based system development.

**Component substitution for system evolution.** In the life-time of a component-based system, especially a large-scale one, replacing a component in the system with a new component would be the most practical and economical way of evolving the system for whatever the purpose, e.g., for better performance, increased capability or even cost effectiveness. In such scenarios, we need to be sure about the substitutability between the components concerned. The information contained in the components' interface specifications allows us to reason about whether or not the existing component can be replaced by the new one regarding their syntactic and structural aspects, their semantic behaviour, their interaction protocols and their quality properties in the context of the system architecture. For example, the new component may provide better performance, but may not have compatible interaction protocols for it to work in the system architecture. This suggests investigation into whether or not an adaptor can be used to overcome the incompatibility [25]. A new component may provide more functionalities for a system to take advantage of (in place of a component currently in use), but may not be usable because it does not satisfy the security requirements of the system (i.e., having no security characterisation or weaker security properties than the existing component).

**Component retrieval and discovery.** As they are developed, software components and systems can be organised in libraries or repositories, and even published on the Internet for Internet-based marketing. Potential users can search the library, repository or the Internet to locate components meeting their specific needs. A central issue of this application scenario is component retrieval and discovery in the sense that the search result should return suitable components relative to the requirements. It is obvious that when stored with the components, the rich interface specifications can play a vital role in the search and discovery process.

An particular application of rich interface specification is dynamic component discovery and system assembly at run-time, over a corporate intranet or the Internet. Imagine the situation where a running scientific literature broking system searches the Internet for literature in a particular subject area and encounters a newly published literature service site/system (a component to be used). To utilise the literature site, the broking system needs to understand the specific services the site offers, the ways in which the site need to be interacted with, and maybe the service quality of the site especially when there are charges related to connection time. The broking system can retrieve this information from the interface specification of the site component/system for analysis. After deciding to use the site, the broking system can connect to the site and carry out the necessary literature search according to the site's interaction requirement. Now the site component has been dynamically assembled with the broking system to form a larger whole. The availability of the interface specification of the site component is one of the key issues for realising this dynamic component discovery and system assembly capability. In fact, the interface specification

can be made available through component introspection or reflection, a technique for embodying component information in the component itself in a retrievable manner [23].

**Infrastructure and tool support.** The above application scenarios suggest the need for infrastructural support, including design-time and run-time tools and facilities. For example, design-time tools can be developed for checking the validity and consistency of interface specifications, and for reasoning about system properties and component substitutability. Run-time support can be based on current middleware technologies. But additional facilities and enhancements are required to support the interoperation of independent components, to monitor and enforce interaction protocols between components, and to achieve run-time component discovery and system assembly.

Currently, we are carrying out a study of enhancing CORBA so that it can handle the rich interface specification and coordinate component interaction according to the interaction constraints. Another study being undertaken aims at utilising and extending the introspection mechanism of JavaBeans for dynamic component discovery and system assembly.

## 5 Related work

The framework for rich interface specification introduced in this article addresses the following aspects of component interface: syntactic signature, structural configurations, semantic behaviour, interaction protocols and quality properties. In addition, use-context-based configuration specification and constraint-based protocol specification are two key specific contributions of the framework. We note that a preliminary version of the framework was presented in [9].

Many have realised the need to provide richer interface specifications for software components or objects than just the syntactic signature. However, most tend to focus on specific aspects. In particular, we note that the current proposal for CORBA components still do not address the issues of specifying interface behaviour and interaction constraints [18].

Beugnard *et al* [3] looked at component interfaces from a contract point of view and outlined a four level model: basic, behavioural, synchronisation and quality of service. While similar to our own framework, this contracts model does not address required services or structural configurations. It is only a high-level classification model without introducing a systematic contracts specification notation.

Borgida and Devanbu [4] investigated using description logics to specify certain behavioural and interaction semantics of component interface as extensions to the current commercial interface definition languages. The work emphasises decidability and static checking of semantic specifications. It has shown a few examples of applying description logics to semantic interface specification in a theoretical sense, but did not discuss how the specification notation can be consistently integrated with IDLs.

Bastide *et al* [2] proposed to use Petri nets to specify the behaviour of CORBA objects, including operation semantics and interaction protocols to a certain degree. However, the work is proposed as a tool for the development of CORBA objects, not necessarily for only interface specification. As such, the specification does not clearly distinguish internal semantics from external observable behaviour of objects.

Our notion of interface configuration structures a component's interface elements relative to use context, where the component may play a number of roles (ports) relative to its neighbouring components and perspectives. The traditional concept of multiple interfaces for a component or object, e.g., in Java or CORBA IDL, provides a rudimentary mechanism for interface organisation. The claim that a component or object implements two interfaces does not necessarily suggest that the two interfaces serve any particular purpose or that there is any necessary relationship between the interfaces (e.g., active at the same time). As such, there is no concept of configuration.

The concept of component roles are introduced in many of the architecture description languages (ADLs) in one way or another. Darwin’s services are similar to our component roles [14, 15]. But a Darwin service can only be one-directional (required or provided), rather than a bidirectional communication connection point. Rapide also has a concept of service that groups related elements in a component’s interface [13]. However, the services in a Rapide component interface are not defined based on the component signature. Rather, the elements in the services form the interface signature without overlap. In a sense, Rapide’s interface structuring mechanism is similar to that of Java. A component in Wright has ports as interaction points where interaction events or operation invocations happen [1]. The interface signature are implicitly embedded in the ports’ CSP-style specifications. Darwin and Wright do not have an explicit concept of interface signature, and they directly focus on services or ports, probably because of their orientation towards architecture description. Rapide deals directly with interface elements, but services (groups of elements) take a rather secondary role. None of these ADLs support multiple configurations.

In our approach, the specification of interaction protocols takes the form of constraints on the relevant interface elements. The constraints can be specified in an incremental manner, i.e. there is no requirement of full protocol specification. Yellin and Strom [25] proposed a state machine-based approach to protocol specification. The interaction protocols of a component are expressed in terms of abstract states and state transitions (interface messages). Since there is no concept of use context in the approach, the specification does not distinguish configuration-independent and configuration-dependent protocol requirements. We also note that the abstract states are not part of the interface signature, and do not have immediate meaning to the user. In Wright [1, 21], the interaction protocols of a component are specified in the form of behaviour specifications of its ports and the computation specification coordinating the ports’ behaviour, all in CSP style. In contrast to constraint-based specification, both the state machine-based specification of the Yellin and Strom approach and the process algebra-based specification of Wright tend to require complete protocol specification, through the consideration of either all the state transition or events sequencing scenarios.

In Rapide, interaction protocols are specified in a combination of algebraic constraints and pattern constraints [13]. Its constraints specification language is much more complex and powerful because it is used for the specification of external interaction protocols and internal computation behaviour of a component. As mentioned earlier, Borgida and Devanbu [4] proposed to use description logics, and Bastide *et al* [2] proposed to use Petri nets, to specify certain types of component interaction protocols.

The notation for the specification of interaction protocols in our approach is very simple. The key constructs are the few temporal relational operators: **proceeds**, **before** and **leadsTo**. They bear some resemblance to the relevant operators of temporal logics such as those in TLA [12], Unity [5] and STeP [16]. However, our temporal operators are about timing relationship of actions, rather than truth relationship of temporal logic formulas. In fact, we plan to formulate a formal semantics for our notation based on some temporal logic.

We mentioned earlier that the constraint-based interaction protocols specification will be likely used more often than the element behaviour specification because the constraint specification is simple and incremental. Our use of constraints for “light-weight” semantic specification has its roots in our earlier work on specifying and managing the structural properties and consistency of software documents [8], and is further inspired by Perry’s work on Inscape [19, 20].

## 6 Conclusions

Rich interface specification for software components represents a major challenge in advancing the current state-of-the-practice of component based software engineering. This article takes a step towards meeting this challenge by providing a framework for rich interface specification. The

framework is comprehensive and yet flexible in that it addresses all the interface aspects of syntax, structure, semantics, interaction and quality, but does not dictate the full specification of all the aspects for a given component. This should make it easy to learn and apply in real-world industrial projects. In fact, the framework was originated from an industrial project.

Another major contribution of our framework is its interface organisation scheme. This scheme recognises that a component may be used in different types of application contexts, and in each context it may play distinctive roles relative to its neighbouring components and from various perspectives. The role-based interface configuration specification reflects this need, and provides a simple and direct way of capturing the context and role information of a component. In particular, multiple configurations reflecting multiple use contexts can be specified before hand or incrementally as they are discovered.

The constraint-based specification of interaction protocols is particularly useful in moving towards a more semantic oriented interface specification for software components. First, it has a more restricted scope and less complex than full behavioural semantic specification. Second, it allows flexible and incremental specification of interaction protocols. This leads to a simple notation that is easy for practitioners to learn and use. In addition, the interaction constraints can be statically checked or dynamically enforced to eliminate many of possible errors in using a component.

While emphasising the simplicity, clarity and practicality for industrial use, our interface specification framework also have the necessary rigour for formal analysis. It provides the necessary basis for us to check formally the consistency of interface specifications, to reason about whether a component meets the requirements of a specific use context, to decide on whether one component can be replaced with another in a given system, and to deduce system properties from component properties. Furthermore, the simplicity and rigour of the framework makes significant contribution towards behaviour-based component discovery and system assembly at run-time.

As the telecommunications software project at Fujitsu Australia has shown, the framework already provides methodological and notational support for component based software development. We are currently investigating infrastructural and tool support for the framework, including the aforementioned investigations into CORBA-based rich interface support and Java-based dynamic component discovery and system assembly. The development of a characterisation and composition model for the security properties of components and systems is also on-going, which is to be integrated with the interface specification framework. Another directly related area of work is architecture-directed system assembly and design that utilises the rich component interface specifications.

## References

- [1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
- [2] R. Bastide, O. Sy, and P. Palanque. Formal specification and prototyping of CORBA systems. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 474–494, Lisbon, Portugal, June 1999.
- [3] A. Beugnard, J.-M. Jezequel, N. Plouzeau, and D. Watkins. Making components contract aware. *IEEE Computer*, 32(7):38–45, July 1999.
- [4] A. Borgida and P. Devanbu. Adding more “DL” to IDL: Towards more knowledgeable component inter-operability. In *Proceedings of the 21th International Conference on Software Engineering*, pages 378–387, Los Angeles, USA, May 1999.
- [5] K.M. Chandy and J. Misra. *Parallel Program Design*. Addison-Wesley, Reading, MA, USA, 1988.
- [6] D. D’Souza and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

- [7] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.
- [8] J. Han. Software documents, their relationships and properties. In *Proceedings of the 1994 Asia-Pacific Software Engineering Conference*, pages 102–111, Tokyo, Japan, December 1994. IEEE Computer Society Press.
- [9] J. Han. A comprehensive interface definition framework for software components. In *Proceedings of the 1998 Asia-Pacific Software Engineering Conference*, pages 110–117, Taipei, Taiwan, December 1998. IEEE Computer Society.
- [10] J. Han and Y. Zheng. Security characterisation and integrity assurance for software components and component-based systems. In *Proceedings of the 1998 Australasian Workshop on Software Architectures*, pages 83–89, Melbourne, Australia, November 1998.
- [11] D. Krieger and R.M. Adler. The emergence of distributed component platforms. *IEEE Computer*, 31(3):43–53, March 1998.
- [12] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [13] D.C. Luckham et al. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–335, April 1995.
- [14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, Barcelona, Spain, September 1995. Springer.
- [15] J. Magee, N. Dulay, and J. Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering*, 1(5):304–312, September 1994.
- [16] Z. Manna and the STeP group. STeP: The Stanford temporal prover (educational release), user's manual. Technical Report STAN-CS-TR-95-1562, Computer Science Department, Stanford University, Stanford, USA, November 1995.
- [17] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.
- [18] OMG. *The CORBA Components: Joint Revised Submission*. OMG TC Document, [http://www.omg.org/techprocess/meetings/schedule/CORBA\\_Component\\_Model\\_RFP.html](http://www.omg.org/techprocess/meetings/schedule/CORBA_Component_Model_RFP.html), 1999.
- [19] D. Perry. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–12, Pittsburgh, USA, May 1989.
- [20] D. Perry. Software evolution and 'light' semantics. In *Proceedings of the 21th International Conference on Software Engineering*, pages 587–550, Los Angeles, USA, May 1999.
- [21] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.
- [22] B. Spiznagel and D. Garlan. Architecture-based performance analysis. In *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, San Francisco, USA, June 1998.
- [23] P. Sridharan. *JavaBeans: Developer's Resource*. Prentice Hall, Upper Saddle River, NJ, USA, 1997.
- [24] J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison-Wesley, Reading, MA, USA, 1999.
- [25] D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.