A Comprehensive Interface Definition Framework 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

Component based software engineering (CBSE) represents an exciting and promising paradigm for software development, attracting much interest and effort from industrial practice and scientific research. Software components are one of the key issues in CBSE. While practical, the current component models are limited in providing support for CBSE. In this paper, we introduce a framework aimed at comprehensive interface modelling for software components in the context of CBSE. This framework deals with interface signature, interface constraints, interface packaging and configurations, and non-functional properties of software components.

1. Introduction

Component based software engineering (CBSE) has great potential to overcome many problems that the object oriented technology has failed to address *adequately*, such as reusability and adaptability. We have seen a strong industrial push (both technological and managerial) for CBSE [5]. For effective and wide-spread adoption of CBSE, there are still many issues that need to be resolved. In general, we need methodology and tool support, organisational and management models, and software component commerce practice [2]. As the basis of methodology and tool support, the following are the three major aspects of concern: components, architectures and infrastructures. For a software *component*, what it provides and requires needs to be clearly defined in order for it to be used and even traded with confidence. The relative independence of software components is a key to the success of CBSE. A component is usually used in the context of a system as part of the system's *architecture*. The relationship between the component and the system should be clearly understood to ascertain the system behaviour. The components used to build a system may be written in different languages, and run across different platforms at different locations. To facilitate the management, execution and communication between the components, *infrastructure* support is required, such as that provided by CORBA, Java RMI and DCOM.

In this paper, we focus on software components, especially their interfaces. The interface of a component defines the component, and serves as the basis for the component's understanding, use and implementation. In principle, the component interface should be the only definitive source for understanding the component, and in many cases may be the only source available (e.g., the case where the component's design and source code are not available). As such, the description of the component interface should be definitive and as comprehensive as possible, to achieve its relative independence. Without clear characterization and definition of component interface, much effort is wasted on the comprehension and adaptation of components each time they are used. Worse still, it may prevent the use of the components. This has been evident in practice. Furthermore, comprehensive interface definition for components is crucial to the management of components as required by systematic component based software engineering.

The component interface definitions in the current industrial component models (JavaBeans and COM components) mainly focus on the syntactic aspect of what the component provides and possibly requires. There is much to be desired for in achieving comprehensive component definition through interface. In general, the interface definition of a component should include what the component provides and requires (i.e., properties, operations and events), the ways in which the component is supposed to be interacted with, the possible roles that the component would play in systems, and the non-functional properties that the component possesses. In this paper, we outline why and how these aspects of a component are to be addressed.

The paper is organised as follows. In section 2, we review some related work. In section 3, we overview an architectural framework for telecommunications software, as a source of examples throughout the paper. In section 4, we introduce a model for comprehensive component interface definition. The key issues addressed include interface sig-

nature, interface constraints, role-based interface packaging and configurations, and non-functional properties.

2. Related work

Existing work relevant to the characterization of software components includes objects in the object-oriented approach, components in software architecture research, and industrial software component models.

Objects. To a certain degree, component based software engineering grows out of object-oriented software engineering. It is useful to look at the characterization of objects in object-oriented software engineering. Object technology has been widely adopted, and has become the standard for much of the software industry. Objects are the core concept of object technology. In general, objects have static and dynamic aspects with structural and behavioural definitions. In terms of characterization, objects take on different forms at different levels of abstraction. In object oriented programming, objects are characterized by *attributes* and *operations*. All the constraints about the object structure and interaction are supposedly embedded or realised in the operations. OMG's object Interface Definition Language also follows this characterization [11]. In some approaches [10], assertions are used to state object invariants (or constraints).

In object-oriented analysis and design, the characterization of objects is enriched to capture the *sequencing and interaction* of object manipulation [3]. The sequencing specification of object operations usually takes the form of a state transition diagram for the object (or class). The interactions of objects are set in the context of scenarios involving a number of objects, and are usually specified in object diagrams and interaction diagrams. In such characterization of objects, the constraints about the object behaviour are captured explicitly through sequencing and interaction specifications, which contribute towards the comprehension, management and use of objects in system development.

Architectural components. Over the past few years, we have seen growing research interest in software architectures [13] and software architecture description languages (SADLs), with efforts like Rapide [7], Darwin [8], UniCon [12] and Wright [1]. In this context, the architecture of a software system is described in terms of components, connectors, configurations, constraints, and possibly other aspects. The treatment of components takes rather diverse forms with varied emphasis. In [9], components are summarized as having the following aspects: interface, types, semantics, constraints and evolution. However, this does not give a clear characterization for effective component management and use.

It is our view that components in software architectures should be looked at as independent entities and in terms of their usage and interactions with other components in the system. As such, an architectural component can be characterized as having *attributes*, *operations* and *constraints*. The attributes are the structural elements of the architectural component, and are usually those that are externally observable. The operations are those allowable on the architectural component. The constraints are those parts of the architecture description that constrain the external usage/interaction and internal composition/state of the component. Different notations are used to describe constraints in existing approaches. Some focus on the constraints on component usage/interaction, while others focus on the component's internal consistency. The constraints that serve as the internal invariants of a component are often not separated from restrictions on its external interaction scenarios.

The current SADLs focus on the system architectural aspect. In contrast, their support for the encapsulation role of components and their relationships to the overall system architecture is relatively weak. Consequently, component characterization is not defined as clearly or comprehensively in the SADLs as one would hope.

Industrial component models. It is pleasing to see that some industry-based component models are being actively pursued and used, including COM components, JavaBeans, initial component model submissions to OMG, and W3C's initiatives on using the Web for component based applications. While the OMG and W3C work represents still-emerging standards, Sun's JavaBeans [15] and Microsoft's COM components [4] are the two market leaders. A good comparative review of these models can be found in [6]. In general, these standards have made CBSE practical. However the support for CBSE is still very much limited, and there is much to be desired for. For example, COM components do not provide much support for events. Some key features of components in JavaBeans are only enabled through naming conventions, which are obscure and do not provide the *right* level of support for system developers. Besides, there is little semantic support available in these models.

3. An architecture for telecommunications software

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

We have seen many telecommunications software systems developed to provide switching, transmission and access management services. Common to these systems is

a particular system architecture framework. This architecture framework involves different kinds of system modules. One system module is the *central manager* (CM) that monitors and coordinates all other modules in a system. Another class of modules are the different types of *managing and service* (MS) units, with each responsible for a specific aspect of the system's functionality. The third class of modules are the *service arbitrators* (SAs) that not only have functionality of their own like MSs, but also manage the services provided by a certain group of service modules. A *service module* (SM) provides some service and is managed by a service arbitrator.

To allow dynamic evolution, upgrade and configuration, all the system modules contain *meta-data* describing themselves, and communicate it to the CM when the modules first come into service in the system, so that the CM is aware of the units' existence and is able to manage their activities. An MS module is relatively independent and can communicate to the CM about its meta-data and interact with other system modules 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 present to the rest of the system, the services provided by the service modules under its control.

Besides, the behaviour of the system and its components (CM, MSs, SAs and SMs) is depending 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 a specific system architecture according to this framework, much simplified for the purpose of discussion in this paper. 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.

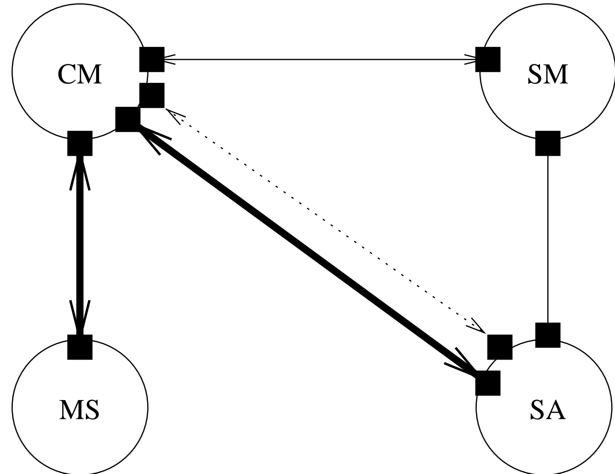

Figure 1. A simplified telecommunications system architecture.

4. A model for comprehensive component interface definition

Proper characterization of software components is essential to their effective management and use in the context of component based software engineering. While there have been industrial and experimental projects that build systems from (existing) components, the approaches taken are ad hoc and heavily rely on the specifics of the systems and components concerned. That is, component based system development is still very much in its infancy, and there are no proven systematic approaches to follow. Characterization of components through comprehensive interface definition is a step towards such systematic approaches and their enabling technologies. In this section, we introduce a model for comprehensive component interface characterization, in the hope to provide a basis for the development, management and use of components.

Figure 2 shows the overall structure for component interface characterization. At the bottom level, 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 for such interaction (i.e., properties, operations and events). The next level up are the *constraints* on the component signature that must be satisfied for the proper use of the component. The component signature and its constraints define the overall capability of the component. The third level concerns the *packaging* of the interface signature according to the component's roles in given scenarios of use, so that the component interface has different *configurations* depending on the use contexts. The fourth aspect of component interface is about the characterization of the component in terms of their *non-functional properties* (code named *illities* [16]). The non-functional properties occupy a special place in this component interface structure, and may interact with the interface signature and configurations. In the following subsections we discuss these aspects of component interface in detail.

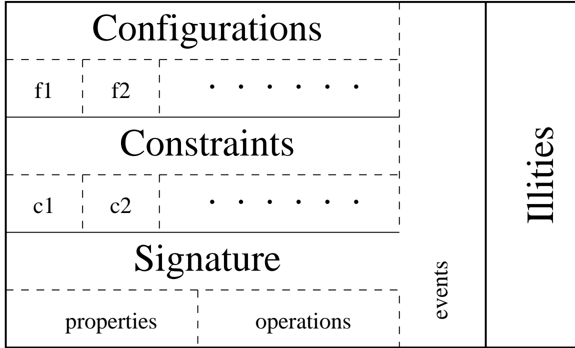

Figure 2. Structure of component interface.

4.1. Interface signature

Fundamental to a component's interface is its signature that characterizes its functionality. The component interface signature forms the basis of all other aspects of the component interface. As commonly recognised, the interface signature of a component comprises *properties*, *operations* and *events*.

A software component may have a number of properties externally observable. These properties 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 observe and even change the property values, to understand and influence the component's behaviour. A common use of component properties is to customise and configure the component at the time of use [14]. It should be noted that certain component properties can only be observed, but not changed.

Another aspect of a component signature is the operations, with which the component interacts with the outside world. The operations capture the dynamic behavioural capability of the component, and represent the service/functionality that the component provides.

Besides proactive control, another form of control used to realise system behaviour is reactive control. 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, and there may be none or many responses to an event from other components in the system.

In the telecommunications software example, the CM component provides a range of services. When another system module comes into service, the CM component first requests the module's signature (i.e., meta-data), then initialises the module by setting its alarm and performance attributes. 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 attribute status. Note that the performance attribute initialisation and status reporting of service modules is done through their proxies. Besides, the CM component can raise a system testing event and all other modules will respond by lighting their testing LEDs.

```
COMPONENT CM {
  SIGNATURE {
    PROPERTIES

    OPERATIONS
      rep_sig(IN Module m, IN Sig sig);
      rep_alm(IN Module m, IN Alarm a);
      rep_perf(IN Module m, IN Perf p);
    EVENTS
      e_system_testing;
  }
}
```

Note that unlike the `e_system_testing` event here, events in general may carry with it a number of value parameters for use by the respondents.

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

```
COMPONENT MS {
  SIGNATURE {
    PROPERTIES
      BOOLEAN enabled;
    OPERATIONS
      req_sig;
      set_alm(IN Alarm a);
      set_perf(IN Perf p);
      system_testing();
    EVENTS
  }
}
```

The SA module has functionality similar to MS, but without the performance attributes. In addition, it manages the SM module's performance attributes 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 configuration. The CM module communicates to the SA the configuration information and sets its protection attributes.

```
COMPONENT SA {
  SIGNATURE {
    PROPERTIES
      BOOLEAN enabled;
      Topology config;
```

```
    OPERATIONS
      req_sig;
      set_alm(IN Alarm a);
      set_ring(IN Ring_protect prot);
      set_pTp(IN PtP_protect prot);
      system_testing();
      set_pxy_perf(IN SM sm,IN Perf p);
    EVENTS
  }
}
```

The SM module's performance attributes are managed by SA and represented by a `performance` property in SM.

```
COMPONENT SM {
  SIGNATURE {
    PROPERTIES
      BOOLEAN enable;
      Perf performance;
    OPERATIONS
      req_sig;
      set_alm(IN Alarm a);
      system_testing();
    EVENTS
  }
}
```

4.2. Interface constraints

The signature of a component interface only spells out the individual elements of the component for interaction in mostly syntactic terms. In addition to the constraints imposed by their associated types, the properties and operations of a component interface may be subject to further constraints regarding their *use*. In general, there are two types of such constraints: those on individual elements and those concerning the relationships among the elements. Examples of the first type are the definition of the operation semantics (say, in terms of pre-/post-conditions) and further range constraints on properties. There is a variety of constraints of the second type. For example, different properties may be inter-related in terms of their value settings. An operation can only be invoked when a specific property value is in a given range. One operation has to be immediately invoked after another operation's invocation.

The explicit specification of these constraints are important. First of all, they form part of the defining characteristics of the component. They make more precise the capability of the component. Furthermore, it is essential for the component user to understand these constraints, so that it can be used properly. Without such constraints, the proper understanding and use of the component will be much harder, and informal, incomplete and imprecise documentations have to be relied on. Worse still, the interface definition of a component may well be the only available source of information about the component.

As an example, we consider two constraints on the MS component. The first states that at the start of the module, it is disabled. The second states that the attribute initialisation operations must be performed before the module is enabled.

```
COMPONENT MS {
  SIGNATURE { ... }
  CONSTRAINTS
    ^(enabled == FALSE);
    (set_alm,set_perf) ..> enabled=TRUE;
  }
}
```

Note that ^ indicates the start of the component's lifecycle, and "`..>`" means "proceeds". The SA and SM modules have similar constraints.

4.3. Interface packaging and configurations

The signature and constraints of a component define the overall capability of the component. For the component to be used, certain packaging is required. It involves two aspects: (1) the component plays different roles in a given context, and (2) the component may be used in different types of context.

In a particular use scenario, a component usually interacts with a number of other components, and plays specific roles relative to them. The interactions between the component concerned and these other components may differ depending on the components and their related perspectives. When interacting with a particular type of component from a specific perceptive, for example, only certain properties are visible, only some operations are applicable, and some further constraints on properties and operations may apply. More specifically, the value range of a property may be further restricted in a particular role. The port specifications in Wright [1] goes some way towards constraining operations at given roles. In general, this suggests the need for defining perspective/role-oriented interaction protocols for a given component, i.e., *multi-interfaces* or *an interface configuration*, as the effect of interface packaging. Since the role-based configuration definition is oriented towards component interaction, a role-based interface of a component should include not only what the component provides but also what it requires from the other end of the interaction (i.e., another component).

The scenarios provide the contexts of use for component. A component may be used in different scenarios and has different role partitions in these scenarios. Therefore, a component may have different sets of interaction protocols, with each set for a scenario in which the component is to be used. This suggests that a component may have different

interface configurations. In principle, an interface configuration should be defined in terms of both the component and the use scenario, and it relates the component to the use context.

When a component is designed, the designer usually has one or more use scenarios in mind. Therefore, a few packaging configurations may be defined for the component interface. When a new use scenario is discovered, a new packaging configuration may be defined.

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

In the reference example, the CM component plays four roles: `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. The `MS management` role defines the interface for interaction between CM and MS from CM's perspective. It has a `PROVIDE` section including the relevant CM signature elements (excluding events), a `REQUIRE` section including elements that should be provided by the other end of the role (i.e., MS), and a `SUBJECT_TO` section stating further constraints on the role.

```
ROLE MS_mnt {
  PROVIDE
    rep_sig(IN MS m, IN Sig sig);
    rep_alm(IN MS m, IN Alarm a);
    rep_perf(IN MS m, IN Perf p);
  REQUIRE
    BOOLEAN enabled;
    req_sig;
    set_alm(IN Alarm a);
    set_perf(IN Perf p);
  SUBJECT_TO
    ^req_sig -> rep_sig;
    (enabled) :> (rep_alm, rep_perf);
}
```

Note that the parameter type `Module` in the operations is further specialised into `MS` in this role, and the role has two constraints. The first states that the first operation on the role is `req_sig` and is immediately followed by the `rep_sig` operation. The second constraint states that the two reporting operations can only be invoked when MS is enabled. Also note that ^ indicates the start of the role, `->` means "immediately proceeds", and "`:>`" stands for precondition.

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

```
ROLE SA_mnt {
  PROVIDE
    rep_sig(IN SA m, IN Sig sig);
    rep_alm(IN SA m, IN Alarm a);
  REQUIRE
    BOOLEAN enabled;
    Topology config;
    req_sig;
    set_alm(IN Alarm a);
    set_ring(IN Ring_protect prot);
  SUBJECT_TO
    ^req_sig -> rep_sig;
    (enabled) :> (rep_alm);
    enabled => (config == RING);
}
```

Note that on this role, there are no performance related operations due to the nature of SA. An additional constraint states that when SA is enabled, its `config` value is `RING`, where `=>` stands for "implies". Besides, the role has the `set_ring` operation in the `REQUIRE` section.

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

```
ROLE SM_mnt_pxy {
  PROVIDE
    rep_perf(IN SM sm, IN Perf p);
  REQUIRE
    set_pxy_perf(IN Perf p);
  SUBJECT_TO
}
```

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

```
ROLE SM_mnt {
  PROVIDE
    rep_sig(IN SM m, IN Sig sig);
    rep_alm(IN SM m, IN Alarm a);
  REQUIRE
    BOOLEAN enabled;
    req_sig;
    set_alm(IN Alarm a);
  SUBJECT_TO
    ^req_sig -> rep_sig;
    (enabled) :> (rep_alm);
}
```

Note that `performance` property of SM is not in this role. Rather, it is related to the interactions between SA and SM.

With the above four roles, the CM component with the ring configuration will have the following definition with some configuration level constraints:

```
COMPONENT CM {
  SIGNATURE ...
  CONSTRAINTS ...
  CONFIGURATION ring {
    ROLE MS_mnt { ... }
    ROLE SA_mnt { ... }
    ROLE SM_mnt_pxy { ... }
    ROLE SM_mnt { ... }
    SUBJECT_TO
      SM_mnt.rep_sig ..> SM_mnt_pxy;
      (SM_mnt_pxy.set_pxy_perf) ..>
        SM_mnt.enabled = TRUE;
      (SM_mnt.enabled) :>
        (SM_mnt_pxy.rep_perf);
  }
}
```

The first constraint states that all activities on the `SM management proxy` role must happen after the `rep_sig` operation in the `SM management` role. The second constraint indicates that the `set_pxy_perf` operation in the `SM management proxy` role must be carried out before the SM module is enabled. The third constraint states that the `rep_perf` operation in the `SM management proxy` role can only be carried out after the SM module is enabled. Notice that these constraints are about the dependencies between roles.

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 `PROVIDE` and `REQUIRE` sections are swapped.

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 those of CM. In the `SM operational management` role, the `performance` property (of SM) is made available to SA for direct management.

```
COMPONENT SA {
  SIGNATURE ...
  CONSTRAINTS ...
  CONFIGURATION ring {
    ROLE SA_mnt { ... }
    ROLE SM_mnt_pxy { ... }
    ROLE SM_operational_mnt {
      PROVIDE
      REQUIRE
        Perf performance;
      SUBJECT_TO
    }
    SUBJECT_TO ...
  }
}
```

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.

The above component configuration is 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 although the types and number of roles do not change in this case. 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:

```
ROLE SA_mnt {
  PROVIDE
    rep_sig(IN SA m, IN Sig sig);
    rep_alm(IN SA m, IN Alarm a);
  REQUIRE
    BOOLEAN enabled;
    Topology config;
    req_sig;
    set_alm(IN Alarm a);
    set_pTp(IN PtP_protect prot);
  SUBJECT_TO
    ^req_sig -> rep_sig;
    (enabled) :> (rep_alm);
    enabled => (config == PtP);
}
```

Note the changes related to the point-to-point setting. Then, CM would have two configurations:

```
COMPONENT CM {
  SIGNATURE ...
  CONSTRAINTS ...
  CONFIGURATION ring { ... }
  CONFIGURATION pTp { ... }
}
```

The same applies to the SA component.

4.4. Non-functional properties (illities)

Another aspect of a component is its non-functional properties, such as security, performance and reliability. In the context of building systems from existing components, the characterization of the components' illities and their impact on the 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 the various illity models in the context of software components and composition. The interface definition framework proposed in this paper can be extended to accommodate these new models.

4.5. Summary

To summarize the above characterization of component interface, we have

Component = signature (properties+operations+events)
+ constraints + configurations + illities.

The properties, operation and events form the signature of the component interface. The constraints further restrict and make precise the definition and hence usage of the component interface. The signature and the constraints characterize the component capability. Each of the configurations is based on a use scenario, and defines a specialised usage of the component. It identifies the roles and defines the role-based interfaces of the component in the given context of use. The component's non-functional properties are useful in assessing the component's usability in given situations and in analysing properties of the enclosing systems.

It is important to distinguish the usage-independent constraints on the component signature from the usage-dependent and scenario/role-based constraints in interface configurations. The former defines characteristics of the component capability. The latter further specialises the component capability in the context of given use scenarios. The constraints in one configuration may not apply in another configuration. However, all the configuration constraints should be compatible with and conform to the usage-independent constraints.

The ability of being able to define different scenario-based interface configurations is important in achieving maximal reuse of components and supporting component customisation and specialisation. A component can be used in different scenarios with clear role assignments by defining corresponding interface configurations. A component can be customised or spelialised through an interface configuration to resolve certain mismatches when being used in a specific system.

5. Conclusions

In this paper, we have introduced a framework for characterizing software components in a comprehensive manner. According to this framework, a component's interface should be defined in terms of its signature (properties, operations and events), constraints, configurations and non-functional properties. The usage-independent constraints and usage-dependent role-based configurations provide the basis for proper development, use and management of software components. The characterization of components' non-functional properties is essential for component selection and system analysis and prediction.

Component based software engineering involves many issues that need to be addressed. On the basis of the work presented in this paper, we are currently investigating the security characterization of components, and working on a supporting tool for the proposed framework. Further issues for investigation include characterization models for other non-functional properties, the compositional (or architectural) aspect of CBSE, and the component and system relationships with underlying infrastructures.

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] M. Aoyama. New age of software development: How component-based software engineering changes the way of software development. In *Proceedings of International Workshop on Component-Based Software Engineering*, Kyoto, Japan, April 1998.

[3] G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition, 1994.

[4] D. Box. *Essential COM*. Addison Wesley, 1998.

[5] CBSE98. *Proceedings of International Workshop on Component-Based Software Engineering*. Kyoto, Japan, April 1998.

[6] D. Krieger and R. Adler. The emergence of distributed component platforms. *IEEE Computer*, 31(3):43–53, March 1998.

[7] D. Luckham et al. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–335, April 1995.

[8] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of 5th European Software Engineering Conference (ESEC'95)*, Barcelona, Spain, September 1995. Springer.

[9] N. Medvidovic and R. Taylor. A framework for classifying and comparing software architecture description languages. In *Proceedings of 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of software Engineering (LNCS -1301)*, Zurich, Switzerland, September 1997. Springer.

[10] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.

[11] OMG. *OMG Documentation*. OMG, http://www.omg.org/, 1998.

[12] M. Shaw et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.

[13] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[14] P. Sridharan. *JavaBeans: Developer's Resource*. Prentice Hall, Upper Saddle River, NJ, USA, 1997.

[15] Sun Microsystems. *JavaBeans API Specification 1.01*. Sun Microsystems, Mountain View, CA, USA, 1997.

[16] C. Thompson. *Workshop on Compositional Software Architectures: Workshop Report*. http://www.objs.com/workshops/ws9801/report.html, Monterey, USA, January 1998.