Semantic and Usage Packaging for Software Components

Jun Han

Peninsula School of Computing and Information Technology
Monash University, Melbourne, Australia**

Abstract. Current models for software components have made component-based software engineering practical. However, these models are limited in the sense that their support for the specification of software components primarily deals with syntactic issues. To facilitate interoperability and proper use of software components, more comprehensive specification of these components is required. In this paper, we present an approach to software component packaging aimed at comprehensive component specification, especially its support for semantic and usage specification.

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 continuously adjust and improve their business practices to 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 vital.

We have seen examples of integrating software components or packages into systems to achieve specific business objectives of enterprises. In general, however, it is not 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 [3]. The mismatches are largely due to the fact that the capability of the components are not clearly described or understood through their *interfaces*. Most commercially available software components are delivered in binary form. We have to rely on the components' interface description to understand their *exact* capability. Even with the components' development documentation available, people would certainly prefer or can only afford to explore their interface descriptions rather than digesting their development details. Furthermore, interface descriptions in natural languages do not provide the level of precision required for component understanding, and therefore have resulted in the above mentioned mismatches. When discovering components and assembling systems at run-time, it becomes a must that the components have precise and even comprehensive interface descriptions.

** Currently on leave in the Department of Computer Science, University College London, Gower Street, London, WC1E 6BT, UK. e-mail: j.han@cs.ucl.ac.uk

Most current approaches for component interface definition deal with primarily syntactic issues, like those of the CORBA Interface Definition Language (IDL). To gain a clear understanding of a component's exact capability, other essential aspects of the component should also be described, including the semantics of the interface elements, their relationships, the assumed contexts of use, and the quality attributes. Our approach to software component specification deals with these various aspects. It has been developed and applied during a large-scale telecommunications R&D project at a multi-national company.

In the following section, we first give a brief account of related work. Then we present an overview of our approach to software component specification before focusing on the semantic and usage specification of software components with examples drawn from an telecommunications application. Such specification provides a basis for proper use and interoperation of software components. Finally, we discuss some further issues before concluding the paper.

2 Related work

Some existing work related to the characterization or packaging of software components includes objects in the object-oriented approach, components in software architecture research, and industrial software component models.

Objects. Software components are closely related to objects in the object oriented approach. 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 follows this characterization [7]. In some approaches [6], 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. 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 contributes towards the comprehension, management and use of objects in system development.

Architectural components. Over the past few years, we have seen growing research interests in software architectures [8] and software architecture description languages (SADLs), with efforts like Rapide, Darwin, UniCon and Wright. In this context, the architecture of a software system is described in terms of components, connectors, configurations, constraints, and possibly other aspects. The components may in turn have their own internal architectures. The treatment of components in this context takes rather diverse forms with varied emphasis. In [5], 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, the architectural components can be characterized in terms of attributes, operations and constraints. The constraints are those parts of the architecture description that constrain the usage/interaction and internal composition/state of the component. Some approaches 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.

Industrial component models. Some industry-based component models are being actively pursued and/or used, including COM components, JavaBeans, component model submissions to OMG, and W3C's initiatives on using the Web for component-based applications. A good comparative review about these technologies can be found in [4]. In general, these standards have made component based software engineering (CBSE) practical. However the support for CBSE is still very much limited. The specification of component interfaces in these standards deals with syntactic issues only.

3 An overview of the approach

As argued earlier, 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. Characterization of components through comprehensive interface specification is a step towards systematic approaches to CBSE and their enabling technologies.

Our approach to component specification aims to provide a basis for the development, management and use of components. It has four aspects (see Figure 1). First, there is the signature of the component, which forms the basis for the component's interaction with the outside world and specifies the necessary syntactic mechanisms for such interaction (i.e., properties, operations and events). The next aspect of component specification concerns the semantics of the component interaction, including the semantic specification of individual signature elements and more importantly additional constraints regarding their interaction protocol. The component signature and its semantic specification define the overall capability of the component. The third aspect of component specification concerns the configurations of the interface signature according to the component's roles in given scenarios of use. In general, the component interface may have different configurations with additional interaction constraints, depending on the use contexts. The fourth aspect of component specification is about the characterization of the component's non-functional properties or quality attributes, such as performance, reliability and security. Due to space limitation, this last aspect will not be discussed any further in this paper.

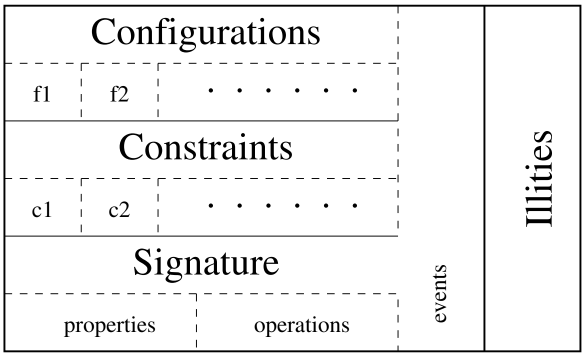

Fig. 1. Structure of Component Interface

4 Interface signature

The interface signature of a component defines in mostly syntactic terms the elements with which the outside world interacts with the component. As popularised by CORBA IDL, JavaBeans and COM, the signature elements include *properties*, *operations* and *events*. In our framework, the interface signature covers all these aspects. In particular, the properties are provided for both design-time and run-time customisation of the component.

To provide a basis for the discussion of semantic and usage packaging in the following sections, let us consider the interface signatures of some components or modules in a typical telecommunications application. One system module is the *central manager* (CM) that monitors and coordinates all other modules in the system. When another system module comes into service, the CM component first requests the module's signature, then initialises the module by setting its alarm and performance attributes (among others). After that, the CM will enable the module for normal service, and the module will then be able to report to the CM about its attribute status (among other information). In addition, the CM component can raise a system testing event and all other modules will respond by lighting their testing LEDs. The following presents part of the CM's interface signature:

```
COMPONENT CM {
  SIGNATURE {
    PROPERTIES
    OPERATIONS
      report_signature(IN Module m_id, IN Sig sig);
      report_alarm_attri(IN Module m_id, IN Alarm alarm);
      report_perf_attri(IN Module m_id, IN Perf perf);
    EVENTS
      e_system_testing; /* all modules light LEDs */
  }
}
```

Note that events in general 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.

Another type of modules in the telecommunications application are *service arbitrators* (SAs) that not only have functionality of their own but also manage in a proxy role the services provided by a certain group of *service modules* (SMs), such as the SMs' performance attributes.

```
COMPONENT SA {
  SIGNATURE {
    PROPERTIES
      BOOLEAN enabled;
    OPERATIONS
      request_signature;
      set_alarm_attri(IN Alarm alarm);
      set_perf_attri(IN Perf perf);
      system_testing();
      set_proxy_perf_attri(IN SM sm_id, IN Perf perf);
    EVENTS
  }
}
```

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

```
COMPONENT SM {
  SIGNATURE {
    PROPERTIES
      BOOLEAN enabled;
      Perf performance;
    OPERATIONS
      request_signature;
      set_alarm_attri(IN Alarm alarm);
      system_testing();
    EVENTS
  }
}
```

5 Interface semantics

The external or interface semantics of a software component can be captured by two types of constraints on its signature elements: 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 are 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 upon the completion of another operation.

The explicit specification of a component's interface semantics is important. First of all, it forms part of the defining characteristics of the component. It makes precise the capability of the component. Furthermore, it is essential for the user of the component to understand the semantic constraints. Only then, proper use of the component can be achieved and therefore the composed system's behaviour is predictable. Without these constraints, the understanding and use of the component will be much harder.

The use of pre-/post-conditions for defining operation semantics has been well studied, such as those used in Eiffel [6] and Catalysis [2]. Here, we focus on constraints concerning the relationships among signature elements. As an example, let us consider two of the constraints on the SA component. The first states that at the start of the module, it is disabled. The second states that the alarm attribute and performance initialisation operation must be performed before the module is enabled.

```
COMPONENT SA {
  SIGNATURE { ... }
  CONSTRAINTS
    ^(enabled == FALSE);
    (set_alarm_attri, set_perf_attri) ..> enabled = TRUE;
  }
}
```

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

6 Interface configurations

The signature and semantic constraints of a component define the overall capability of the component. For the component to be used, certain further 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 contexts.

In a particular use scenario, a component usually interacts with a number of other components, and plays specific roles relative to each of 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. For example, the port specification 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., *an interface configuration*. 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).

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 sets of interaction protocols, with each set for a scenario. 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 concerned, and it relates the component to the use context. Usually, when a component is designed, the designer 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 later, a new packaging configuration may be added.

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

```
ROLE SA_management {
  PROVIDE
    report_signature(IN SA m_id, IN Sig sig);
    report_alarm_attri(IN SA m_id, IN Alarm alarm);
    report_perf_attri(IN SA m_id, IN Perf perf);
    e_system_testing;
  REQUIRE
    BOOLEAN enabled;
    request_signature;
    set_alarm_attri(IN Alarm alarm);
    set_perf_attri(IN Perf perf);
    system_testing();
  SUBJECT_TO
    ^request_signature -> report_signature;
    (enabled) :> (report_alarm_attri,report_perf_attri);
    e_system_testing -> system_testing;
}
```

Note that the parameter type `Module` of the operations is further specialised into `SA` in this role, and the role has three constraints. Th first states that the first operation on the role is `request_signature` and is immediately followed by the `report_signature` operation. The second constraint states that the alarm and performance reporting operations can be invoked only when SA is enabled. The third constraint states that the `e_system_testing` event is to be immediately responded by the `system_testing` operation. Also note that ^ indicates the start of the role, `->` means "immediately proceeds", and "`:>`" stands for precondition. In general, these constraints concern the interaction between the two components relative to the role.

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

```
ROLE SM_management_proxy {
  PROVIDE
    report_perf_attri(IN SM sm_id, IN Perf perf);
```

```
  REQUIRE
    set_proxy_perf_attri(IN Perf perf);
  SUBJECT_TO
}
```

The SM management role of CM is as follows:

```
ROLE SM_management {
  PROVIDE
    report_signature(IN SM m_id, IN Sig sig);
    report_alarm_attri(IN SM m_id, IN Alarm alarm);
    e_system_testing;
  REQUIRE
    BOOLEAN enabled;
    request_signature;
    set_alarm_attri(IN Alarm alarm);
    system_testing();
  SUBJECT_TO
    ^request_signature -> report_signature;
    (enabled) :> (report_alarm_attri);
    e_system_testing -> system_testing;
}
```

Note that `performance` property of SM is not relevant to this role, and it is done through the SM management proxy role. The CM configuration should capture the relationships between the SM management proxy role and the SM management role regarding the management of SM's performance attributes.

```
COMPONENT CM {
  SIGNATURE ...
  CONSTRAINTS ...
  CONFIGURATION ring {
    ROLE SA_management { ... }
    ROLE SM_management_proxy { ... }
    ROLE SM_management { ... }
    SUBJECT_TO /* configuration constraints */
      SM_management.report_signature ..>
        SM_management_proxy;
      (SM_management_proxy.set_proxy_perf_attri) ..>
        SM_management.enabled = TRUE;
      (SM_management.enabled) :>
        (SM_management_proxy.report_perf_attri);
  }
}
```

The first constraint states that all activities on the SM management proxy role must happen after the `report_signature` operation in the SM management role. The second constraint indicates that the `set_proxy_perf_attri` operation in the

`SM management proxy` role must be carried out before the SM module is enabled. The third constraint states that the `report_perf_attri` 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 interactions between roles.

7 Further issues

Architecture-directed system composition/assembly. The composition/assembly of a system from components is another important issue. At the time of system composition, the relevant configurations of the used components are selected, the roles of the configurations are linked/connected to each other, and the properties, operations and events of corresponding roles are mapped.

Our approach to rich component specification facilitates the derivation of a system's properties from those of its components (as specified in their interfaces) and those of the system architecture. However, this requires further research into composition models of various system properties. In addition, component substitutability can be studied using the information available in the component interfaces.

Infrastructural support. To facilitate the rich component packaging as described in this paper and support system composition and execution, infrastructural support is required, 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 are required to support the interoperation of independent components, and especially the run-time component discovery and system assembly.

Introspection. A major motivation for having rich and formal component specification is to facilitate the embodiment of component knowledge in the component itself, termed *introspection*, so that independent or run-time analysis of the component may take place with appropriate infrastructural support. The component specification approach introduced in this paper supports component introspection in the sense that all the interface information can be subject to introspection.

8 Summary

On the basis of the interface signature, the interface semantics of a component restricts and makes precise the definition (and hence the usage) of the component interface. In particular, the interface constraints capture the relationships among the elements of the interface signature, and characterise the protocol about how these elements are to be used together. The interface configurations are based on the component use scenarios, and define specialised usages of the component in terms of the roles that the component plays in a given scenario. The role-based interface partitioning and associated role and configuration constraints clarify the component's capability in a given context of use.

It is important to distinguish the usage-independent semantic 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 or conform to the usage-independent constraints.

The industrial project where our component packaging framework has been developed and applied, concerns the development of a telecommunications access network system involving software and hardware co-design. Combined with object oriented analysis techniques such as scenario analysis, the framework had been used in the architecture design of the system. Immediate benefits of using this framework have been the clear definition of the subsystems/modules' capabilities through their interfaces, the clear identification of the interactions between the modules, and the analysis of system behaviour at architectural level. This has significantly reduced the interaction between the teams responsible for the various modules, and avoided many of the architectural changes later in the development cycle that had been experienced in earlier projects.

It is evident that semantic and usage specifications allow better support for the reuse and interoperation of software components. With precise interface definition, the match and replacement of a component with another component is also made easier. We are currently investigating the infrastructural support needed for semantic and usage packaging of software components, especially in situations where software components are dynamically discovered and assembled into target systems at run-time.

Acknowledgement. I would like to thank my colleagues at Monash University and University College London for their useful comments. The anonymous reviewers also helped to improve the paper's presentation.

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. D. D'Souza and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1999.
3. 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.
4. D. Krieger and R.M. Adler. The emergence of distributed component platforms. *IEEE Computer*, 31(3):43–53, March 1998.
5. N. Medvidovic and R.N. Taylor. A framework for classifying and comparing software architecture description languages. In *Proceedings of ESEC/FSE'97 (LNCS -1301)*, Zurich, Switzerland, September 1997. Springer.
6. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 1997.
7. OMG. *OMG Documentation*. OMG, http://www.omg.org/, 1998.
8. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Upper Saddle River, NJ, USA, 1996.