An Approach to Software Component Specification

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 characterization/specification of software components primarily deals with syntactic issues. To avoid mismatch and misuse of components, more comprehensive specification of software components is required, especially in a scenario where components are dynamically discovered and used at run-time over corporate intranets and the Internet. Our approach to software component specification aims at comprehensive interface modelling/packaging for software components. It deals with the semantic, usage, quality as well as syntactic aspects of software component specification.

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. 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, in general, 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 [Garlan et al, 1995]. The hard-to-identify 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 over corporate intranets and the Internet, it becomes a must that the components have precise and even comprehensive interface descriptions.

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 aspects through comprehensive interface specification. It has been developed and applied during a large-scale telecommunications R&D project at a multi-national company. It provides not only the basis of notational and tool support for software component specification, but also the basis of methodological guidance for architecture-directed and component-based system development, composition and integration.

While our framework highlights comprehensive packaging, it is unrealistic to expect that every component is to be defined formally and comprehensively in practice. For example, JavaBeans and COM components are still very useful even though their interface definitions are mainly syntactic. In these cases, the full understanding of the components has to rely on other means, e.g., informal documents. Or, the user of the component is happy with the partial information that has been offered by the interface. It is very important to allow such flexibility in packaging software components. It is the component user who decides whether the provided interface information is enough to warrant the adoption of the component in his/her use context. This is particularly true for the quality attributes of the component. We generally refer to this flexibility as "sliding characterization/specification".

An analysis of existing industrial component models and the need for comprehensive component characterization can be found in [Han, 1998a]. In the following sections, we give a brief account of our approach to software component specification. Further details can be found in [Han, 1998b; Han and Zheng, 1998].

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. That is, component-based software engineering is still very much in its infancy. 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. First, there is the *signature* of the component, which forms the basis for the component’s interaction with the outside world and deals with the syntactic aspect of the necessary 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* on the component signature in terms of their proper use. The component signature and its semantic constraints define the overall capability of the component. The third aspect of component specification 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 specification is about the characterization of the component in terms of their *non-functional properties* or *quality attributes* (code named *illities* [Thompson, 1998]).

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 use (i.e., observe and even change) their values, to understand and influence the component's behaviour. A common use of component properties is for component customisation and configuration at the time of use. 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 outside world interacts with the component. The operations capture the dynamic behavioural capability of the component, and represent the service/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, and gives the system the capability of dynamic configuration.

In our approach, the specification of interface signature takes a form similar to the current Interface Definition Languages found in CORBA and Java, including assemble-time and run-time properties, operations and events.

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 a number of further semantic 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 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 after another operation's invocation.

The explicit specification of semantic properties are important. First of all, they form part of the defining characteristics of the component. They make more precise about the capability of the component. Furthermore, it is essential for the user of the component to understand these constraints. Only then, proper use of the component can be guaranteed and therefore the composed system's behaviour is predictable. Without such constraints, the proper understanding and use of the component will be much harder. Informal and possibly incomplete and imprecise documentation has to be relied on. While we all know the dangers and problems associated with such a scenario, it has even greater significance for component based software engineering. This is because the interface definition of a component may well be the only source of information for the component as we may not have access to its source code or any other development documentation, e.g., in a scenario where components are discovered and used dynamically at run-time.

The use of pre-/post-conditions for defining operation semantics has been well studied, such as those used in Eiffel [Meyer, 1997] and Catalysis [D'Souza and Wills, 1998]. Our approach focuses on constraints concerning the relationships among signature elements. A common example in telecommunications systems is that a system module has to be initialised through a sequence of operations before it is enabled to accept normal requests (e.g., invocations of further operations). Specific mechanisms are available to specify this type of constraints in our approach.

Interface packaging and configurations

The signature and the semantic 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 contexts. 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 perspective, for example, only certain properties are visible, only some operations are applicable and some further constraints on properties and operations may apply. More specifically, for example, the value range of a property may be further restricted in a particular role. In general, this suggests the need for defining perspective/role-oriented interaction protocols for a given component, i.e., *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 (another component) of the interaction.

Scenarios provide the contexts of use for a 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 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.

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, a new packaging configuration may be added. It should be noted that the packaging of an interface configuration is subject to the component's underlying capability as defined by the component's signature and semantic constraints and will not alter this defined capability.

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 our approach, mechanisms are provided for specifying interface configurations and roles within configurations. Additional constraints about component interaction can be specified in association with roles and configurations.

Quality attributes (illities)

Another aspect of a component is its non-functional properties or quality attributes, 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 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 the various illity models in the context of software components and composition. For a particular quality attribute, we need to address two issues: (1) how to characterize the quality attribute for a component, and (2) how to analyse the component property's 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 characterization of the quality attribute will change in different contexts of use. Currently, we are investigating the security properties of software components and their impact on system composition in the context of developing distributed electronic commerce systems [Han and Zheng, 1998]. In general, the interface definition of component illity characterization will be dependent on the specific characterization models developed. While we do not have definite models available yet, the component specification framework proposed in our approach can be extended to accommodate new models concerning quality attributes.

Summary

In our approach to software component specification, the properties, operation and events form the signature of the component interface. The constraints further restrict and make precise the definition (and hence the usage) of the component interface. The signature and the constraints characterize the component capability. The configurations are based on the component use scenarios, and define specialised usages of the component. A configuration identifies the roles and defines the role-based interfaces of the component in a given use context. 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. In general, the proposed framework provides the basis of notational and tool support for component-based system development, composition and integration. It also contributes to the standardisation of software component specification and its infrastructural support.

Our approach to component specification has been developed and applied in the context of a real-world industrial project that concerns the development of a telecommunications access network system involving software and hardware codesign. Combined with object oriented analysis techniques such as scenario analysis, the approach had been used in the system's architecture design. 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.

References

1. D. D'Souza and A.C. Wills, 1998. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley.
2. D. Garlan, R. Allen and J. Ockerbloom, 1995. Architectural Mismatch: Why reuse is so hard. *IEEE Software,* 12(6): 17-26.
3. J. Han, 1998a. Characterization of components. In *Proceedings of 1998 International Workshop on Component-Based Software Engineering,* Kyoto, Japan, pages 39-42.
4. J. Han, 1998b. A Comprehensive interface definition framework for software components. In *Proceedings of 1998 Asia-Pacific Software Engineering Conference,* Taipei, Taiwan, Pages 110-117. IEEE Computer Society Press.
5. J. Han and Y. Zheng, 1998. Security Characterisation and integrity assurance for software components and component-based systems. In *Proceedings of 1998 Australiasian Workshop on Software Architectures,* Melbourne, Australia, pages 83-89.
6. B. Meyer, 1997. *Object-Oriented Software Construction,* 2nd edition. Prentice Hall.
7. C. Thompson. *Workshop on Compositional Software Architectures: Workshop Report.* http://www.objs.com/workshops/ws9801/report.html, Monterey, USA.