JBCDL: An Object-Oriented Component Description Language*

Wu Qiong, Chang Jichuan, Mei Hong, Yang Fuqing
(Department of Computer Science & Technology, Peking University, Beijing 100871)

Abstract

This paper introduces Jade Bird Component Description Language (JBCDL) which is a part of Jade Bird Component Library (JBCL). JBCDL is based on Jade Bird Component Model (JBCOM). JBCOM is a 3C-based hierarchic component model that is composed of specification and implementation layers and with uniformity and self-contained composition. The main purpose of JBCDL is to describe component interface. It mainly applies to component composition under the help of (semi-) automatic tools. JBCDL has the following features: 1) code-wares and design-wares that adopt object-oriented paradigm as the description objects; 2) adopting object-oriented paradigm itself; 3) uniformly describing components of different forms (such as class, framework, and etc.); 4) integrating well with Jade Bird Component Library.

1. Introduction

The research on component description and composition can be traced back to the 'module' proposed by Parnas in 1970's [14]. Early research efforts mainly focused on module interconnection languages (MILs), such as MIL75 [4], Intercol [15], and etc. In 1980's, the research direction turned to component description languages (CDLs). The most representative works included OBJ [6] and LIL [7] developed by Gougen, ACT TWO[1] developed by "Berlin approach," Meld [11], and etc. Litvintchouk and Mastsumoto argued that the difference between these two kinds of languages mainly lies in that the MIL level description is declarative, while that of CDL is imperative [12]. In 1990's, most efforts are spent on how to introduce the virtues of MILs into CDLs, which means to enable CDLs to describe component as well as component sub-system. Main works include Π [3], CDL [5], CIDER [18], LILEANNA [17], RESOLVE [2], OOMIL [8], and etc.

JBCDL is a part of Jade Bird Component Library (JBCL). JBCL saves all kinds of software development results -- from different development phrases, with different forms and representations -- into component library, and provides tools to help end-users to find the needed components. The development of JBCL is a part of National Key Project "Industrialization software production technology and its supporting system."

The main purpose of JBCDL is to describe component interface. It can be applied in the following three directions: 1) component composition under the help of (semi-) automatic tools; 2) component verification based on the formal information in component interface; 3) component retrieval based on specification matching techniques. The objective of JBCDL is to fulfil the above three directions at the same time, but the current version of JBCDL is mainly designed and implemented to fulfil the first direction. However, we have considered about the potentiality of future extensions. JBCDL has the following features: 1) code-wares and design-wares that adopt object-oriented paradigm as the objects of description; 2) adopting object-oriented paradigm itself; 3) uniformly describing components of different forms (such as class, framework, and etc.); 4)

* This work is sponsored by the national key project in State 9th Five-Year Plan and National 863 High-Technology Project.

0-8186-8551-4/98 $10.00 © 1998 IEEE

integrating well with Jade Bird Component Library.

This paper introduces the main ideas of JBCDL, including Jade Bird Component Model and the syntax structure of JBCDL. More details of JBCOM and JBCDL can be found in [JadeBird Project Group 97A, JadeBird Project Group 97B]. JBCOM, as the foundation of JBCDL, mainly elucidates what kind of components can be described and what kind of attributes they should have. JBCOM will be discussed in section 2. Section 3 presents the syntax structure of JBCDL and two examples.

2. Jade Bird Component Model

Jade Bird Component Model (JBCOM) is the foundation of JBCDL, and the kernel of the conceptual model of component library. In this section, we first define what kind of components JBCOM can describe. Then several important design rules of JBCOM are discussed. In the last part of this section, JBCOM itself is introduced.

2.1 Objects of description

Software reuse can be divided into *direct reuse* and *indirect reuse* due to different component attributes:

1. *Indirect reuse* refers to the reuse of components that contain documented knowledges, such as requirement specifications, design documents, patterns, test plans, and etc. Till now, there is not any formal mechanism that can facilitate the direct compositions of such components. Usually, they are composited manually into target system by adopting following two-step process: a) Developers thoroughly understand the knowledges contained; b) They use these knowledges in the development of the target system. Although the reuse of such non-code components can not benefit software development process by producing an executable application, it will be conducive to the efficiency and quality of software development.
2. *Direct reuse* refers to the reuse of components that can be represented by some kinds of programming language. This kind of components can be composited (semi-) automatically in order to produce an executable application directly.

JBCL saves all kinds of software development results -- from different development phrases (such as analysis, design, coding, test, and etc.), with different forms (such as class, framework, pattern, and etc.) and different representations (such as graph, pseudo-code, programming language, and etc.) -- into component library. JBCOM is a model mainly for those direct-reusable components in the component library, i.e., code-wares and design-wares of classes and frameworks. A framework is a sub-system composed of a group of cooperating classes or abstract classes (and their sub-classes). Compositions of classes and frameworks, or frameworks and frameworks, can also produce frameworks of larger granularity.

2.2 Design rules

We deem that JB_COM must have following features:

1. **Enough expressive ability**: In order to obtain enough expressive ability, JBCOM must comply with 3C model. 3C model [16] is a prescriptive component model that was proposed by Will Tracz on the "Reuse in Practice Workshop" in 1989. In 3C model, a component consists of at least the following three parts: *concept*, *content*, and *context*. The *concept* is the abstract description of what a component does. The *content* is the implementation of the *concept*, and it describes how the component implements the functions that are given in the

concept. The *context* depicts the dependencies between the component and its environment on different levels, and it can be further divided into three parts: 1) *conceptual context*, which depicts the dependencies between the concepts of different components; 2) *operational context*, which depicts the characteristics of the manipulated data; 3) *implementation context*, which describes how the component depends on other components for its implementation. In addition, we have examined some representative CDLs (such as RESOLVE [2], LILEANNA [17], OOMIL [8], CDL [5], and etc), and require JBCOM to have at least the same expressive ability.

2. **Self-contained composition**: To attain the maximum reusability, the component composition should be self-contained, i.e., a composition of components is also a compositable component.
3. **Uniformity**: Components of different forms must have uniform interface, structure, and composition mechanism.

2.3 Model Design

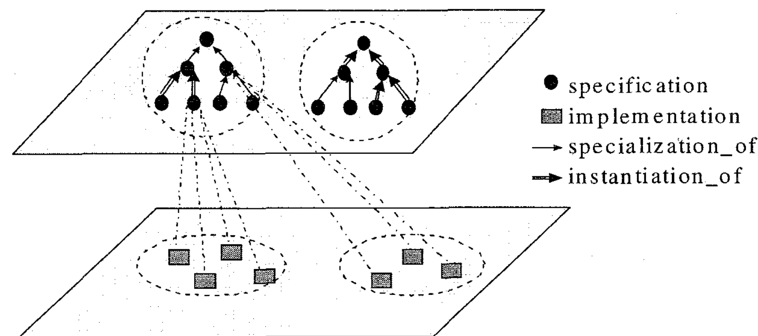

Figure 1. The macro-model of JBCOM

In JBCOM, a component is divided into two parts: specification, including concept, conceptual context and operational context, and implementation, including implementational context and content. Each implementation must correspond to a specification, while a specification may correspond with several implementations. There are some specifications that do not correspond to any implementations. These specifications can be used as simplified components to describe design-wares. The separation of specification and implementation can be viewed as a kind of abstraction that can be used on similar specifications. In general, they are two kinds of similarities among specifications: 1. The functions are same except for the operated data object, such as stack of integer and stack of float; 2. Most functions are same except for a few. For the first case, we can use parameterization to get a more general specification that is called specification template, and instantiates this template to get the specification that operates on special data object. For the second case, we can get a more general specification by only keeping common functions, and create the specialization of this specification by inheritance. Till now, we have educed the macro-model of JBCOM that is illustrated in figure 1. This macro-model is composed of two layers: specification layer and implementation layer. The specification layer is composed of several trees of specifications that are linked by specialization and instantiation relation. Each specification of it can correspond with several implementations.

In JBCOM, a component consists of seven parts as illustrated in figure 2:

1. *Template parameters*, which lists the parameters needed when this component is a specification template;
2. *Provided functions*, which depicts the signatures and semantics of the provided functions. It corresponds to the *concept* in 3C model;
3. *Requirements*, which depicts the required functions of the co-operators using specifications. In the following, we name these specifications as requirement specifications. In JBCOM, the interface of a component is defined as the combination of *Template parameters*, *Provided functions*, and *requirements* parts. Requirement specifications do not refer to concrete components, but only virtual images of co-operators. To reuse this component correctly, reusers must connect these virtual images to existing components that conform with their requirement specifications. Traditional MILs can only perform composition at source code level, which limits the range of the behaviours that can be obtained from composited components [13]. While the introduction of requirement specifications can solve this problem. They depict the required behaviour specifications of the co-operators of a component, and all the implementations that conform with these requirement specifications can composite with this component, which increases the flexibility of composition and enlarges the range of behaviours that can be obtained from composited components;
4. *members*, which lists all the member components that fulfil the provided functions of the component by co-operation;
5. *connections*, which properly connects the components listed in the above two parts to make each component connected to all its co-operators;
6. *imported specifications*, which declares all the imported specifications in the component. It corresponds to the *conceptual context* in 3C model;
7. *implementation*, which describes how the provided functions are implemented. It corresponds to the *implementational context* and *content* in 3C model.

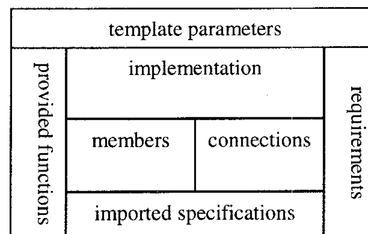

Figure 2. Component structure in JBCOM

In JBCOM, a class is described in the following ways: 1. The *provided functions* part describes the signatures and semantics of its methods; 2. The *members* part describes its attributes; 3. The *requirements* part converts the co-operators that are deeply buried in the implementation into virtual images that are depicted by requirement specifications. To reuse this class correctly, reusers must connect these virtual images with the existing components that conform with their requirement specifications. This conversion enables class implementations not to depend on other components; 4. The *connections* part is empty.

A framework is a sub-system composed of a group of cooperating classes or abstract classes (and their sub-classes). In JBCOM, these classes or abstract classes can be described by using

specifications, and their sub-classes by using specializations of these specifications. Therefore, a framework can be described as a component sub-system composed of several co-operating components. In JBCOM, a component sub-system is still a compositable component, as illustrated in figure 3: Its *provided functions* part depicts the provided functions of this sub-system; Its *members* part lists out the internal components in this sub-system; Its *requirements* part depicts the required interfaces of the external components that co-operate with this sub-system; Its *connections* part properly connects the components listed in *members* and *requirements*.

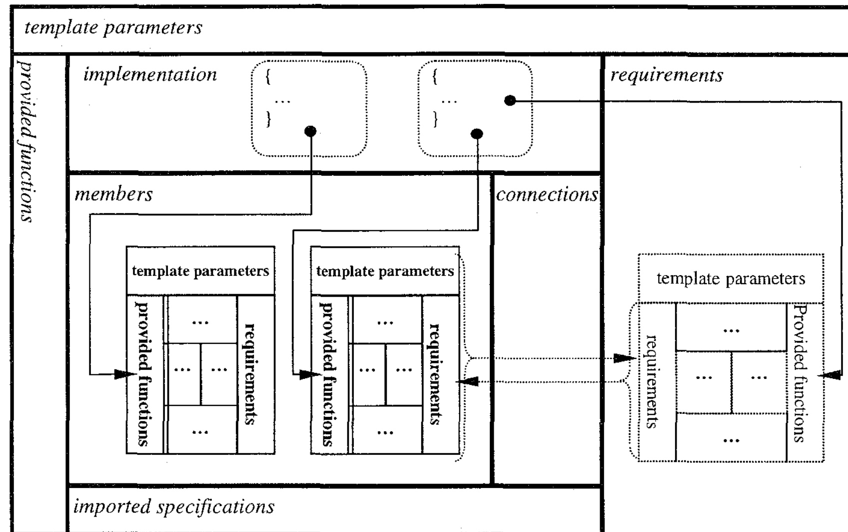

Figure 3. The micro-model of JBCOM

In conclusion, JBCOM is a 3C-based hierarchic component model that is composed of specification and implementation layers and with uniformity and self-contained composition.

3. Jade Bird Component Description Language (JBCDL)

The objective of JBCDL is to fulfil the following three aspects at same time: 1) component composition under the help of (semi-) automatic tools; 2) component verification based on the formal information in component interface; 3) component retrieval based on specification matching techniques. Current version of JBCDL is mainly designed and implemented to fulfil the first aspect. However, we have considered the possible requirements of the following two aspects to keep JBCDL extendable.

JBCDL should be understood as easily as possible. Because, component description is a clue that reusers use to judge whether the component is reusable. The more intelligible the component description is, the higher the possibility is for the component to be correctly reused. So JBCDL adopts natural-langauge-like syntax structure. However, the relation between intelligibility and concision is often a kind of trade-off, i.e., intelligibility of a CDL is often at the cost of its

concision, which makes the developing of components more burden-some. In JBCL system, we solve this problem by providing corresponding tools, such as JBCDL editor.

The syntax structure of JBCDL complies with JBCOM. It describes a specification in 6 parts: *parameters*, *provides*, *requires*, *contains*, *connections* and *imports*, corresponding to *template parameters*, *provided functions*, *requirements*, *members*, *connections*, and *imported specifications* in JBCOM. Implementations are described by using specific programming languages.

In most CDLs, such as RESOLVE [2], CDL [5], and etc., inheritance of component interface is realised implicitly by re-exporting the interface of super-component from the interface of sub-component. Implicit realization of inheritance is flexible, because it does not require implementation language to support inheritance mechanism. However, we consider that inheritance is very important to component understanding, and what's more, most popular OOPLs (such as C++ and Smalltalk) support inheritance. As a result, we decide to realize inheritance of specification explicitly in JBCDL, but we also recognize that un-limited inheritance may cause trouble. In JBCDL, inheritance is restricted to subtype, i.e., sub-specification can only rename the provided functions of its super-specification, or add new functions. Furthermore, JBCDL only support single inheritance. Because in the component library, sub-specification need to inherit all the description information of its super-specification (such as terms and keywords), which means that multi-inheritance may cause the inconsistency of semantics.

Imports part declares all the imported specifications in a component, which is conductive to the intelligibility of component specification. In JBCDL, the instantiation of a specification template is also described by using inheritance. Example 1 illustrates specification template Stack and its instantiation Stack_Of_Integer:

```
specification Stack
  parameters
    Item;
  provides
    push(Item) returns none;
    pop returns Item;
    depth returns integer;
end specification
```

```
specification Stack_Of_Integer inherits
Stack[Integer]
end specification
≡
specification Stack_Of_Integer
provides
    push(Integer) returns none;
    pop returns Integer;
    depth returns integer;
end specification
```

Example 1

Example 2 that is adopted from [8] illustrates a home-heating system, which contains four components: clock, controller, thermometer, and heater. Its working process is as the following: the clock triggers the controller to detect current temperature through the thermometer, and if the current temperature is lower than a given point, it will turn on the heater, or else it will turn off the heater.

```
specification device
  provides
    on returns none;
    off returns none;
end specification

specification heater inherits device
  provides
    state returns integer;
```

```
specification controller inherits triggerable-device
  imports
    sensor, device;
  requires
    sens is a sensor;
    dev is a device;
  provides
    renames trigger as clock;
    read-control-value returns float;
```

```
end specification

specification sensor
  provides
    read returns float;
end specification

specification thermometer inherits sensor
  provides
    renames read as read_temp;
end specification

specification triggerable-device
  provides
    trigger returns none;
end specification

specification clock
  imports
    triggerable-device;
  requires
    tri-dev is a triggerable-device;
  provides
    set-beat (float) returns none;
end specification
```

```
    set-control-value (float) returns none;
end specification

specification home-heating
  imports
    clock, controller, thermometer,heater;
  contains
    control-clock is a clock;
    temp-controller is a controller;
    temp-gauge is a thermometer;
    space-heater is a heater;
  connection
    control-clock . tri-dev = temp-controller;
    temp-controller . sens = temp-gauge;
    temp-controller . dev = space-heater;
  provides
    read-temperature returns float;
    set-temperature (float) returns none;
    set-beat (float) returns none;
end specification
```

Example 2

4. Conclusion

Currently, most CDLs (such as LILIEANNA [17], RESOLVE [2], CDL [5], and etc) are designed to support the reuse of ADA program. Compared with these CDLs, JBCDL is more appropriate to describe components that adopt object-oriented paradigm. Furthermore, the uniformity and self-contained composition of JBCOM enable JBCDL to support composition at different levels, which makes JBCDL more flexible.

In the future, we will continue our research in the following directions: 1) developing JBCDL reverse-engineering, editing and composition tools; 2) adding formal information to JBCDL; 3) discussing how to fulfil component verification based on the above formal information; 4) discussing how to fulfil component retrieval based on specification matching techniques.

Reference

[1] Blum, H. Ehrig, and F. Parisi-Presicce, Algebraic specification of modules and their basic interconnections, Journal of Computer and System Sciences, 34:293-339,1987.

[2] Paolo Bucci, Stephen H. Edwards, etc., Special Feature: Component-Based Software Using RESOLVE, ACM SIGSOFT. Software Engineering Notes, Vol. 19, No. 4, pp.21-67, October 1994.

[3] Cramer, W. Fey, M. Goedicke, and M. Grode-Rhode, Towards a formally based component description language - a foundation for reuse, Structured Programming, 12:91-110, 1991.

[4] Deremer and H. Kron, Programming in the large versus programming in the small, IEEE Transaction on Software Engineering, pp. 321-327, June 1976.

[5] Robert J. Gautier, Huw E. Oliver, Mark Ratcliffe, and Benjamin R. Whittle, CDL--A Component Description Language for Reuse, International Journal of Software Engineering and Knowledge Engineering, Vol. 3, No. 4, pp.

499-518, 1993.

[6] Gougen, Parameterized programming, IEEE Transactions on Software Engineering, SE-10(5):528-543, 1983.

[7] Gougen, Reusing and interconnecting software component, IEEE Computer, pp.16-28, Februray 1986.

[8] Pat Hall and Ray Weedon, Object Oriented Module Interconnection Languages, in Advances in Software Reuse, Selected Papers from the Second International Workshop on Software Reusability, R. Prieto-Diaz and W. B. Frakes eds., pp. 29-38, Lucca, Italy, March 24-26, 1993.

[9] JadeBird Project Group, JadeBird Component Description Language - JBCDL, Technical report, Department of Computer Science and Technology, Peking University, 1997.

[10] JadeBird Project Group, JadeBird Component Model, Technical report, Department of Computer Science and Technology, Peking University, 1997.

[11] Kaiser and D. Garlan, Melding sofwtare systems from reusable building blocks, IEEE Software, 4(4):17-24, July 1987.

[12] Litvintchouk and A. Mastsumoto, Design of Ada systems yielding reusable components: An approach using structured algebraic specification, IEEE Transaction on Software Engineering, SE-10(5):544-551, 1984.

[13] Hafedh Mili, Fatma Mili, and Ali Mili, Reusing Software: Issues and Research Directions, IEEE Transactions on Software Engineering, Vol. 21, No. 6, pp. 528-562, June 1995.

[14] Parnas, On the criteria to be used in decomposing systems into modules, Communication of the ACM, pp. 1053-1058, December 1972.

[15] Tichy, Software Development Control Based on System State Descriptions, PhD Thesis, Carnegie-Mellon University, January 1980.

[16] Tracz W., Implementation working group summary, In James Baldo, ed., Reuse in Practice Workshop, Pittsburgh, Pennsylvania, July 1989

[17] Will Tracz, LILEANNA: A Parameterized Programming Languages, in Advances in Software Reuse, Selected Papers from the Second International Workshop on Software Reusability, R. Prieto-Diaz and W. B. Frakes eds., pp. 66-78, Lucca, Italy, March 24-26, 1993.

[18] Whittle and M. Ratcliffe, Software component interface description for reuse, IEE BCS Software Engineering Journal, 8(6), November 1993.