Component Modeling for System Analysis and Design

Zheying Zhang¹, Matti Rossi²

¹Department of Computer Science and Information Systems
University of Jyväskylä, PL 35, FIN-40351 Jyväskylä, Finland
zhezhan@cc.jyu.fi

²Helsinki School of Economics
P.O. Box 1210, FIN-00101 Helsinki, Finland
mrossi@hkkk.fi

Abstract

Despite the growing acceptance of component thinking in software development, there is inadequate support of component in a (meta)CASE environment. Taking into account the features of components and its involved metaCASE environment, we improve the current component model by adding more supplementary information and offering more flexibility in its interface description. Such a component model and the associated functionality for component classification and retrieval greatly enhance the possibilities of incorporating reuse and components into the early phases of systems development practice.

1 Introduction

Information system development (ISD) is a complex process involving modeling of several aspects of the system: the static structure, the dynamic behavior, and the intended functionality. In order to facilitate this process, different methods for ISD, modeling techniques, and their supporting environments have been proposed by the academia and industry [1]. However, most of the tools and practices have failed to deliver the promised productivity increases. Recently usage of ready-made components in software engineering has received considerable attention [2]. The idea of software components (or software ICs [3]) is to provide software industry with same level of ready-made parts that are available in the hardware fields. It gradually becomes a key for improving time-to-market, productivity, quality and re-use in software development. While component techniques have not matured to the point where users can assemble applications, they are reshaping the ways information systems communities analysis, design, implement, and maintain information systems.

To extend the benefit of component technology into the whole ISD lifecycle, proposals for component reuse in the analysis and design phase have started to emerge [4, 5]. In [5], a component model is proposed from three aspects: concept, content, and context. Such a model can be applied in a configurable CASE (i.e. metaCASE) environment to wrap diverse design artifacts as complete and independent compositions for future reuse. However, due to its generic role of specifying design artifacts into components at each information type level [6], such a model is too abstract to use effectively at the system analysis and design stage. To be really useful the components must form an integrated part of a software development methodology that supports reuse oriented development practices. In order to make components easier to understand and reuse, we propose the use of supplementary information that is defined in the component model and can be used to guide the
selection of components. In this paper we present some ideas of such supplementary information.

In this paper we construct the methodical support for component-based development (CBD) in a metaCASE environment. Our goal is to advance the component technology support for system analysis and design in the metaCASE environment. This will be accomplished by improving the component structure based on the previous work (see [5]), especially the concept part (component interface) which is specifically used at the modeling stages of ISD. It is the first step to further study the facility support of CBD in a metaCASE environment.

2 A component model in a metaCASE environment

We apply the component model within a metaCASE environment, called MetaEdit+, which is a customizable CASE environment that supports both CASE and metaCASE functionality for multiple users. It supports and integrates multiple methods and includes multiple editing tools for diagrams, matrices and tables. All information in MetaEdit+ is stored in the Object Repository, including methods, diagrams, matrices, objects, properties, and even font selections. Hence, modification of system designs (or methods) in one MetaEdit+ client is automatically reflected to other clients on commit, guaranteeing consistent and up-to-date information. A repository is composed of projects, each of which contains a set of graphs that describe a particular system, and possibly some metamodels. The adoption of full object orientation enables flexible organization and reuse of software components in the environment and a high level of interoperability between tools. MetaEdit+ supports Data independence as defined in traditional data base theory i.e. tools operate on design information without “knowledge” of its physical organization, or logical access structure. Representation independence forms a continuum with data independence and it allows conceptual design objects to exist independently of their alternative representations as text, matrix or graphical representation. This principle allows flexible addition of new tools, each one only responsible for its own paradigmatically different view on the same underlying data. Our goal is to supplement the currently available tools with a comprehensive set of tools for reuse support (see [6] for details).

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies. It can be deployed independently and is subject to composition by third parties [7]. In general, it is an executable asset that can be independently developed, delivered, and deployed as a unit into an application. Components for system analysis and design differ from the software components. They are design artifacts defined during the system analysis and design process. These design artifacts are represented in varying notations according to the supported methodology. They become concrete after necessary re-specification and connection to platform specific implementations. For example in UML a component is a very generic wrapper for some part of the system [8]. We believe that this is too simple view for useful components. In [5], we define a component from three perspectives: concept, content, and context. The component metamodel and its representation in UML notation are illustrated in Figure 1.
The concept provides abstract information about a component’s functionality and associated semantics by using a faceted schema, which forms a facet descriptor using a set of predetermined facets and their values to specify component interface. The content forms a chunk of executable code providing a specific service for system implementation, or a diagram specializing a structure, a function, or a state transition for system analysis and design. The context defines the contextual dependencies among components and specifies the “domain of applicability”. It is a complementary part that provides enriched contextual information including definitional dependency; reuse dependency, usage context and the implementation context. The three perspectives represent a component as an organic whole.

3 Component model for the system analysis and design

The component model offers the potential to assemble applications much more rapidly than ever before by reusing existing pre-build components to meet the users’ demands at the early phases of software lifecycle. Meanwhile, any component aiming at being reused should respect two general criteria: completeness and flexibility.

Completeness means that the component must provide all the information necessary to reuse. Karlsson [9] discusses the diverse information that should be presented in the component as follows:

- Classification information to facilitate rapid retrieval of suitable components
- Description to facilitate understanding of the functionality of a component
- Documentation to facilitate understanding how to use it
- Information of its origin to facilitate obtaining support or additional information
- Qualification information to facilitate evaluating and testing it.

Comparing the above with the current component model, the current model lacks adequate documentation to facilitate understanding how to use the component, which should be the essential as the means to specify the contract between a component and any others that may be used in conjunction with it [10]. Moreover, the component concept should offer more pragmatic knowledge about component usage to convey information between component users. For example, it should provide the “access points” and the necessary parameters of a component, and conditions to connect an “access point” with the external artifacts.

Besides, components should respect the requirement of flexibility in order to be reused widely. In the model, we have a facet descriptor for component classification and retrieval, which is normally metamodeled by method engineers based on general domain knowledge. However, in practice, we do not know exactly who (which
project) will create and use components and what information users expect to get from components. Method engineers may ignore the requirements from system designers and provide a faceted schema that was thought reasonable for them in a given development context. The pre-defined faceted schema will thereby limit the utilization of component. In order to promote the component agility, we should seek for solutions to defining a faceted schema that is more user-oriented and flexible to gather for organization or project specific needs.

In order to diffuse the component technology throughout ISD lifecycle, we have to elaborate the component model properly, especially its concept part. Based on the discussion above, an extended component model is presented in Figure 2.

The improved component model expands the concept part with enriched and pragmatic information for reuse, specifies the content as the semantic combination of a collection of design elements, which are defined in terms of the metamodelling paradigm. In MetaEdit+, these design elements are objects, properties, relationships, and roles. Meanwhile, this metamodel also shows the nested nature of components, which means that a larger-grained component can be composed of several finer-gained reusable components. When reusing a component, the user then can make a choice between reusing overall composite component (including all its sub-components) or just one of its finer-gained components.

![Figure 2 Component metamodel in a metaCASE environment](image)

The component concept contains two types of information: component attribute specified in terms of facets, and the interface specification in terms of interface facet.

Similar to the component model stated in section 2, facets are perspectives to specify component properties, like the application domain and the abstraction level. A collection of facets forms a faceted schema that is used for component classification and retrieval. In order to increase the flexibility to specify a component, besides the facets pre-defined by the method engineer, the faceted schema is extensible by the component creator. More domain specific information can be abstracted as facets and added into the component descriptor.

Interfaces are the means by which components connect. Technically, an interface is a set of named specifications that can be invoked by users. As shown in Figure 2, the interface facet offers information about how a component is used. Since a component can be reused in diverse contexts with different interfaces, it has at least...
one interface facet. The interface facet is made up of one port and its related facet information. The port forms the gateway that can be used to connect the component with other components or design elements. The related facets offer knowledge about the connection between the port and the outside world, for example, the possible relationship(s) between the port and the environment, and conditions to use the given port.

The component concept is expanded to provide more practical information and more opportunities for reuse. Therefore, component agility is enhanced to some extent. In the following sections, we will illustrate the improvements with examples from the prototype of improved component model in MetaEdit+.

4 Extensible component facet descriptor

The problem of encoding reuse information is hard. Consider for example the sheer mass of items we need to classify and the number of domains to cover. As commented by the Desperado project [11], there is no way of knowing in advance what reuse information will be useful in the future. If component facets are fully classified in advance, it is often hard to adapt them into the local context of a development organization. On the other hand, it is easier to classify, and especially reuse the components, if the facet schemas are familiar to the developers and to the application domain. To overcome the limitations of fixed facet model, we introduce an extensible facet descriptor into the component model. In this schema a set of predefined facets are given, and the users are free to define new facets and also either limit the possible values for these facets, e.g. explicit definition of value domains through value lists.

As an example we could consider a situation, where a development organization develops mobile phone applications. In this area the components are quite specific and also well known in advance. We believe that this kind of approach is particularly effective in clearly limited domains and for reuse within product families. As the domain is well known, the facets used for description can be much finer grained and the vocabulary used should be well known for the component creators and users. As an example we could have a facet called phone type, which immediately leads into classification into components for GSM, CDMA, GPRS, UMTS and so on. Now, when the users retrieve components of even quite a large granularity, they can use the phone type as the determining factor. In fixed classification schemata this is not possible. Another example is from the prototype implemented in MetaEdit+. As shown in Figure 3, besides the fixed facets like Graph type, Abstraction aspect, and Abstraction level, the component creator qualifies for adding more facets to Store Structure component: new feature and another feature. The value of each facet can be view through the facet pair window.

The system supports the retrieval of components based on user defined facets as well as the fixed facets. As the facet type is just another meta property in the system, it can be applied for any project, where it would be useful. We believe that this kind of extension mechanism greatly enhances the usability of a faceted schema by adding context and organization specific facets for reuse. This in itself is nothing new, but the nature of the metaCASE environment gives users immediately the same tools to manipulate the user defined facets as there are for the fixed facets.

5 Component interface elements

Besides the attributes specified by the extensible component facet descriptor, users are keen on the usage of components in a specific organizational context. In some
contexts it suffices to consider the interface of a component to define the component’s access points [7]. The interface information should be clarified by the component interface element through which the users can access the services provided by the component. It includes port information and the interface facets.

Different from the facet descriptor, the interface facets step up the component usability by defining the port and its behavior in terms of a role, pre-condition, and post-condition. A port defines the access or output point of a component. It could be an object in the component, or even a relationship that connect the component with the externals. A role specifies how the external participates in the component via the specific port. Its value indicates if the port is a plug-point for “input”, “output”, or without any direction; accordingly a right-headed arrow, a left-headed arrow, or a solid pie is attached to the component in its graphic representation. The pre-condition specifies a set of inputs that the component is defined over, called the legal input, and implies the possible relationships between the port and the external design elements. It illustrates the possible types of object that can join in the port through a legal relationship and a role. The post-condition defines the situation that must hold between input and a valid output. If the pre-condition is true when the component is connected with the other, the component is guaranteed to terminate in a state where the post-condition is true. For example, in the MetaEdit+ prototype, as shown in Figure 3, the component name *Store Structure* is presented in the middle of the rectangle. It is reused in the *Product availability* class diagram via its port which is a class named *Store* and shown in the middle of trapezoid, beside the rectangle. There is no specific role definition, a solid pie is thereby attached with the component. Due to the long text description of pre- and post-condition, instead of shown in the graphic representation, they are only shown in the component interface definition window.

![Figure 3 Integrate Component: Store into Class Diagrams: Product availability](image)

Generally, a static component, like a class diagram, describes the structure of the model. It can be divided into several parts down to the level of a single class component, since a class is encapsulated with certain services, which suits for the general component definition. In principle, any class in the class diagram can work as a plug-point. There is no need to specify the role of the port due to the static feature of this kind of component. It thereby has no special demands for the post-condition, like the example shown in Figure 3 that the Port *Store* has no values assigned to the role and post-condition.
However, a dynamic model or a functional model component describes aspects concerning the state changes with time, or processes transformations. It is important to indicate how the component can be integrated with the external during the process of changes or transformations. Therefore, a clear role specification of “input” or “output” is necessary, and the post-conditions should be set to lead the component to its termination. Examples of a state transition diagram component, its interface facet description, and the integration are shown in Figure 4.

Because a component for system analysis and design is generally represented as a graph, it is a “white-box” component and users can “look inside” it to assess whether the component meets their requirements. When creating a component, component creators define the component interface elements. However, it is hard to collect all demands from diverse stakeholders and define a complete interface facet list for use. In order to improve the agility of such a “white-box” component, component users can define the additional ports and its interface facets in case the interface elements are insufficient for use. Figure 4 shows an example of a dynamic model component called ProductStates, which is a state transition diagram. It illustrates the case that the user specializes and reuses the component by adding one more port according to his demand. Component ProductStates specifies the state changes of product in a wholesale system. It has one “input” port called Quality checking defined by its creator. When the component is reused, the user would like to improve it by adding a state in case there are no enough products when the product is gathered. Therefore, one more “output” port Gathering the product with its interface facets is defined by the user. A more complete state transition diagram of product is then composed by connecting the component with one more state through two ports. It is represented at the left bottom in Figure 4.

As component developers and users are sometimes ignorant of each other, the specification of the interface becomes the mediating tool that lets the two parties work together. The purpose to create such an interface facet is to provide necessary information for component understanding, adaptation, and reuse.
6 Conclusion

In this paper, we enhanced the component model by means of metamodelling to better support organization and project specific reuse of assets. The aim was to enhance component concept presentation in a metaCASE environment, so that users can easily understand the component functionality and its usage. We defined the component model and added it in a straightforward way into a metaCASE environment. The main difference between our approach and most other approaches to components is that when components are treated as first class objects within the environment, their use can be supported at repository and tool level within the metaCASE tool. This is a major advantage over most other methods (e.g. Catalysis) which provide no specific tool based support for the components, but rather rely on standard CASE tools such as Rose.

The developed component model serves as a basis for the component reuse process to represent the component features, functionality, access points, and the related syntax and semantics conditions. We demonstrated briefly how this kind of support could enhance a mainstream method, such as UML by integrating component concepts into UML notations. One of our main goals was to use a generic component model, which can be integrated to different methods/techniques. We believe that the methodical support and organization specific flexibility provide should make these methods and tools easy to adopt and deploy. The methods provided make it possible for users to select, understand, and adapt a component into the design models of an application.

We believe that this kind of inclusion of component model and associated functionality for classifying and retrieving components greatly enhances the possibilities of incorporating reuse and components into mainstream systems development practice already during the early phases of systems development. However, due to the limited length of the paper, we did not demonstrate how the improved component model enhances the component classification, retrieval and reuse processes, and the empirical study on component functionality in MetaEdit+. In the future research we seek to define concrete methods and support tools that apply these tools in industrial contexts. Also we need to develop methods, which are engineered from the start to support CBD development.

References


