Defining Components in a MetaCASE Environment

Zheying Zhang

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

Abstract. In this paper we describe how to improve method reusability in a metaCASE environment called MetaEdit+. The suggested component based approach helps unify design artefacts into components with explicit interfaces and meaningful context descriptions. We describe a method artefact from three perspectives: concept, content, and context. We create a component concept by using a hierarchical facet-based schema, and represent contextual relationship types by using definitional and reuse dependency, usage context, and implementation context links. This is the first attempt to explicitly define components into a metaCASE environment.

1 Introduction

Manufacturing industries learnt long ago the benefits of moving from custom development to an assembly of pre-fabricated components to speed up time-to-market and reduce costs. In the same manner, information system development can be significantly improved if applications can be quickly assembled from pre-fabricated software components. This strategy, called component-based development (CBD), has been a goal for nearly two decades. The recent focus on component-based systems has shown that CBD allows better quality, faster development, and effective change management [1]. It represents a further step in the industrialisation of information system development, and will become the next wave in application development.

Introducing components in information system development offers potential to reuse code and other design artefacts, and will lead to an faster assembly of applications. Some component-based CASE tools, e.g. SoftModeler [2], ObjectiF [3], and Rational Rose 98 [4], provide support for component based analysis and design. These tools expand beyond the functionality of current metaCASE environments that were originally designed without component thinking. Because metaCASE environments can be used for both methodology specifications and methodology supported system design activities, they manipulate more diverse artefacts than CASE tools. Most of these artefacts are used as independent task units, and relate to the others by semantic definitions or dependencies. These artefacts and the "knowledge" embedded in form good sources for reuse, especially across projects. It is therefore feasible to introduce component thinking into a metaCASE environment. Based on our knowledge, there are no studies of how to introduce components in environments supporting method engineering. This is the first attempt to study component definition

B. Wangler, L. Bergman (Eds.): CAiSE 2000, LNCS 1789, pp. 340-354, 2000

for all design artefacts. It will bring benefits of CBD into the metaCASE environment. The most significant one is that the application development can be improved by using quickly assembled components [5]. Although some metaCASE environments, e.g. MetaEdit+ [6], provide limited facilities to support reuse activities in the method engineering and software design process, the application of component concept will further improve reuse. In this paper, we will take MetaEdit+ as a target environment of our study, but the same principles apply to other environments as well.

This paper describes a framework for scalable tools and techniques that increase artefact reusability. We study features of design artefacts, and the possibility of packaging them into "components". After that, these artefacts are further defined into components based on three perspectives: concept, content, and context. These perspectives are based on a 3C model [7]. In the process of defining components, possible solutions to component representations in MetaEdit+ are outlined.

2 Components

A component can be thought as a unit of independent deployment that can be reused by a third party. ECOOP'96 (the European Conference on Object-Oriented Programming) offers the following definition of a software component:

"A software component is *a unit of composition* with contractually specified *interfaces* and explicit *context dependencies* only. A software component can be deployed independently and is subject to composition by third parties."

A component thus has three basic features: explicit interfaces, a context of services, and the design for reuse. Among these basic features, explicit interfaces, the context of services, and the unit of composition form a complete component. The design for reuse is the purpose and the result of an application of a component. In industry, a component forms a coherent package of software that can be independently developed and delivered, while in a metaCASE environment, a component can be any design artefact, i.e. a metamodel, or a chunk of source code. The metamodel component is different from the method fragment as discussed in method engineering language (MEL) [8]. A method fragment can be used to describe every aspect of a method. It does not integrate the concept and the relationships among method fragments into a unified whole. Instead the metamodel component has an explicitly specified interface and relationships with the environment. Such a structure can ease the management of metamodels and improve reusability.

A good specification of a component helps users understand it and reuse it [9]. One of the attempts to characterise reusable software components is the 3C model [10]. It provides a metamodel to describe a software component using three distinct aspects of a reusable component: concept, content and context. A concept forms the abstraction captured in a component; a content is the implementation of the interface; and context describes the environment where a component interacts. The component includes the conceptual context, the operational context and the implementational context. We can see that these three aspects form the basic elements and features of software components currently used in industry, e.g. object components in CORBA.

3 Components in MetaEdit+

A metaCASE environment can be used to build system development tools. Instead of providing one fixed software engineering environment, as most traditional CASE (computer aided software engineering) tools do, a metaCASE environment provides also facilities for method engineering which involves design, construction and adaptation of methods, techniques and tools for various system development contingencies [11]. Therefore, a metaCASE tool can generate a variety of CASE tools, depending on the methodology used. Several metaCASE tools have been developed during the last decade, e.g. commercial products such as MetaEdit+ (MetaCASE Consulting), MethodMaker (Mark V), ToolBuilder (Sunderland/IPSYS /Lincoln); and research prototypes such as MetaView (Alberta) and MetaGen (Paris).

MetaEdit+ is a fully configurable metaCASE environment that provides functionality for dual processes: CASE and CAME (computer aided method engineering) [6, 12-15]. MetaEdit+ offers GOPRR [12] metamodelling language to implement the CAME functionality. It provides concepts, rules, and semantics to specify, generate, and customise methodologies as sets of metamodels that can be further instantiated to models that present solutions for an application domain. GOPRR stands for the acronym of Graph, Object, Property, Relationship, and Role. These types form the primary meta data types that can be used to model artefacts for methodologies. The conceptual GOPRR modelling constructs are shown in Table 1.

Table 1. GOPRR Metamodelling language

Meta Data Types	Description
Graph (G)	A graph is a specification of a method (technique). It is an aggregation concept that contains all other GOPRR meta types. E.g. the definition of State Transition Diagram is a graph.
Object (O)	An object is a conceptual thing in the universe around us. E.g. State, Start, and Stop are objects in the definition of a State Transition Diagram.
Property (P)	Properties describe/qualify characteristics associated with other meta types. E.g. State name is a property of State.
Relationship (R)	A relationship forms an association between two or more objects. It connects objects through roles. E.g. Transition is a relationship used to connect two States.
Role (R)	A role is a link between an object and a relationship to specify how an object participates in a relationship. E.g. From and To are two roles indicating that a Transition relationship is from one State to another State.

Based on GOPRR metamodelling language, MetaEdit+ accordingly involves diversity artefacts for methodology specification and information system design. For example, in the methodology specification process, the object *State*, the relationship *Transition*, and other types of data are artefacts that form a larger artefact in the UML methodology called a State Diagram. In the same manner, in the system design process, e.g. a *Phone call state diagram* has different states such as *Busy*, *Dialling*, *DialTone*, and so on, which are independent artefacts with their own features. These

artefacts can be regarded as encapsulated units, but normally without complete interfaces and contextual relationship representations. They are some sorts of "semi-components". We temporarily call them components, although they are not themselves components. In MetaEdit+, what we call components are sets of design "models", represented in graphic "notations"; metamodel artefacts of methodology specifications; and chunks of code that represent services. Meanwhile, due to the symmetric processes supported in MetaEdit+, they differ in terms of information level, granularity, and representational features.

3.1 Information Levels

Artefacts in MetaEdit+ can be grouped into two levels: model level and metamodel level. The model level information covers a project development lifecycle. It consists of design artefacts with different forms of representation for different development stages, such as diagrams for system design and source code for final implementation. Some examples of design artefacts in MetaEdit+ are shown in Fig. 1. It represents a *Phone call state diagram* in the form of diagram, table, and matrix. Besides, State *DialTone* is represented as one composition artefact.

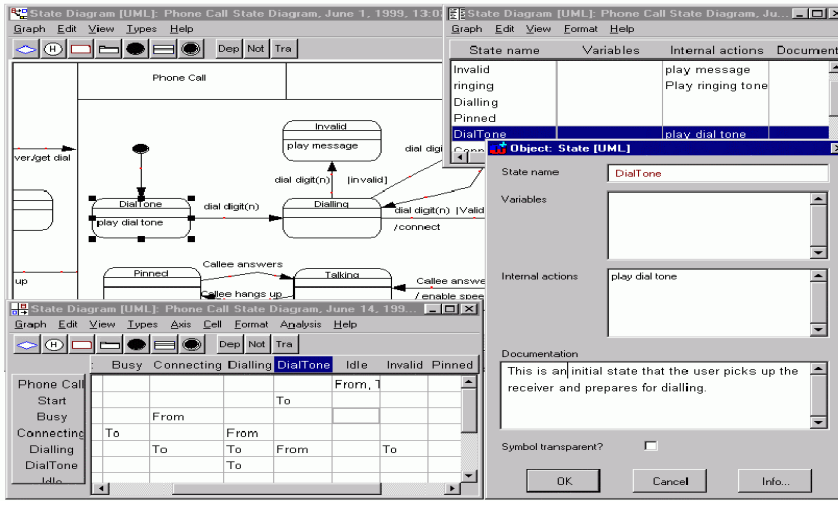

Fig. 1. Model level artefacts to specify a Phone Call State Diagram in MetaEdit+

The metamodel level information forms a semantic specification of system development methodologies by using GOPRR metamodelling language. These metamodels are specified in form-based tools and their graphic representations are specified in the Symbol Editor. As shown in Fig. 2, the Object *State* and its symbol definition, as well as other artefacts and semantics to construct the Graph *State Diagram*, such as binding definition, decomposition definition, and explosion definition, are specified in different windows. *State Diagram* is part of the specification of UML methodology. We can see from Graph Tool that *State Diagram* belongs to a project UML.

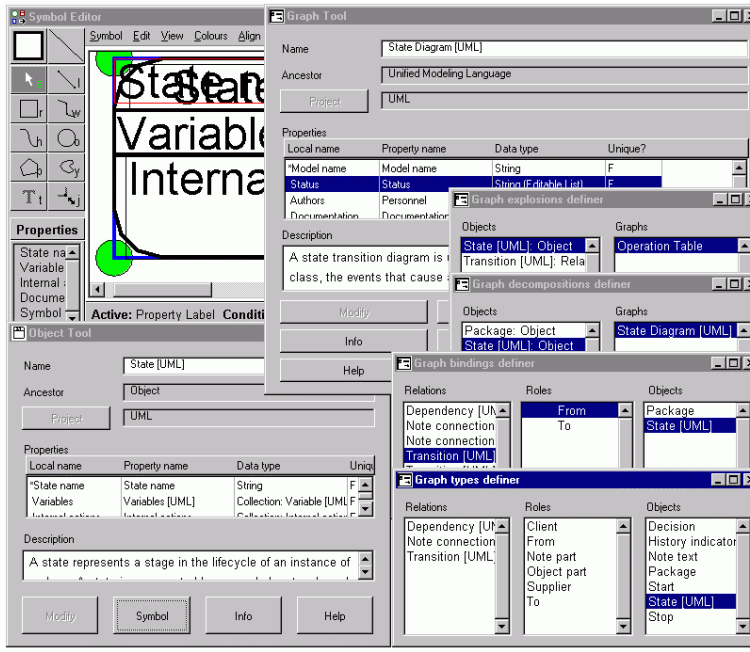

Fig. 2. Form-based tools to specify State Diagram in UML in MetaEdit+

It is known that a methodology provides particular disciplines and techniques to support information systems development. A system development process can not deviate from its supported methodologies. The artefacts on these two information levels thereby have close type-instance relationship: design artefacts on the model level are instances of metamodels. The metamodel provides the type of meta data from which design artefacts are instantiated. For example, the *Phone call state diagram* is an instance of State diagram of methodology UML.

3.2 GOPRR Based Component Granularity

Components in MetaEdit+ have different sizes. A part of the definition of a State *DialTone* can be taken as one component. An instance of a state diagram, e.g. *Phone call state diagram*, can be taken as a component as well. Accordingly, components can be grouped into different granularity levels.

Granularity specifies the size of the "manipulation" or "retrieval" units under consideration. It is derived from representational or operational semantics that is based on underlying modelling concepts and their representations. A fine-grained granularity level component includes a detailed and formal specification, such as a chunk of source code. A coarse-grained component includes more general information about a sub-system, or the whole of an application and hides away the detailed specification and implementation, such as a diagram. The coarse-grained component can be detailed into one or several fine-grained sub-components with a coherent syntax and semantics. Generally, the component granularity implies how

effective CBD can be. As the components become coarser grained, the reuse opportunities decrease, while the productivity increases [5].

In MetaEdit+, the basic meta data types are applied to model various methodologies. These can be further instantiated to specify an information system development project either in part or as a whole. We thereby distinguish between a component unit, a graph, and a project.

- *Component* units are primary data types of MetaEdit+. They are non-property date types, like object, relationship, role, or their instances. For example, *State* in Fig. 2 is an object type component, and *DialTone* in Fig. 1 is an instance of *State* in the project *Phone call design*. Component units form the smallest atomic level component in MetaEdit+. Any further decomposition would not be worth the effort of storage, retrieval, and manipulation.
- A *graph* forms a collection of objects, relationships, roles and properties. It provides a representation of a technique on the metamodel level, and results in diagrams describing specific tasks within a specific application domain. For example, a specification of a state diagram is a graph level component.
- A *project* is a design product, or a plan to produce it. Like graph, it has a dual meaning. On the metamodel level, a project is a methodology including a set of techniques and rules that guide system development, such as UML methodology. On the model level, it is represented as a system development project consisting of design models and code.

4 Describing Components in MetaEdit+

To benefit from CBD, we should define a uniform interface to represent design artefacts in MetaEdit+ and find a reasonable way to display contexts among components, although the original design of MetaEdit+ does not include the component idea. Since 3C model [7] provides basic elements of components used in industry, we apply it to describe components. The model is represented in Fig. 3.

According to the 3C model, a component in MetaEdit+ can be described using three aspects: concept, content, and context.

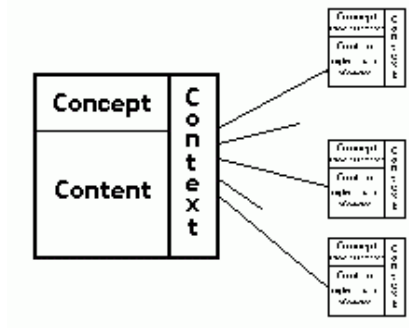

Fig. 3. Component model

The *concept* is realised by an interface specification. It provides abstract information about a component's functionality and associated semantics. Since a component interface defines its access point and indicates its further usage which are concerned by users, the way to specify the interface thereby has an important role in the component representation. *The content* forms an implementation of the concept. It is a substantial part of a component and outlines features and behaviours described in the interface. Following the specification on the concept and the content level, *the context* defines the contextual dependency among components and specifies the "domain of applicability". It is the most beneficial and complicated part of a component specification, since the context provides more detailed information from a wider perspective that helps users understand and reuse a component.

The three aspects represent a software component as an organic whole. Since components are not limited to source code, the representation of a concept, content, and context need to be adapted to the metaCASE environment and associated artefacts. In the following, we will discuss these three composition aspects in the context of MetaEdit+.

4.1 Concept

The component concept provides abstraction information of its functionality and associated semantics, like the pre- or post-conditions. In the same environment, components normally have the same concept schema. However, in MetaEdit+, since different granularity level components represent the information on different abstraction levels, it is difficult to outline a uniform interface. Generally, besides the general information such as the name, author, version and so on, a component unit has no explicit service to be represented in its interface. For example, *DialTone* is a state with information such as the name, the created date, or the version. It is difficult to describe its service although *DialTone* is a component in the *Phone call state diagram* that specialises a call session. On the other hand, the graph level components can express a concrete function and imply the semantics of its usability. Its interface thereby includes properties and explicit functionality specification, such as *Phone call state diagram* that specifies the state transition in a call process. The project level components represent the organisational information and application domain information. Their interface thereby includes a project purpose and its development strategies.

For each granularity level, components have different abstraction schemata to represent their interface. Moreover, since components in the same project have close dependencies, they can share some common conceptual descriptions in the interface. To effectively represent the component concept and to avoid the redundancy, we need a hierarchical concept description schema.

A hierarchical concept description schema forms an extension of a faceted schema, as proposed by Prieto-Díaz [16, 17]. It includes several faceted schemata. A faceted schema uses a number of predetermined perspectives called facets to represent the concept. A facet forms a viewpoint of a component. The viewpoint definition may include functions that components perform, their outputs, the application domain, and so on. A value for a facet is a facet value. The set of facet values for a component

forms a fact descriptor, which represents the component concept. Generally, the facet-based schema works effectively in the repository when the collection of components is very large and growing continuously, and there are larger groups of similar components [18]. This is exactly the form that a MetaEdit+ repository has.

The hierarchical structure between the schemata derives from distinct component concept representations at different granularity levels. The same schema specifies the concept of the same granularity level components, but is distinguished from the schemata on other granularity level components. By using the hierarchical attribute description schema, the integration relationship between a fine-grained sub-component and its interrelated coarse-grained super-components becomes tractable. The hierarchical concept description schema is represented in Fig. 4. The faceted schema of components on both the graph level and the component unit level includes a facet "SuperComp" which is used to trace its coarse-grained super-components and thus to achieve the shared conceptual information.

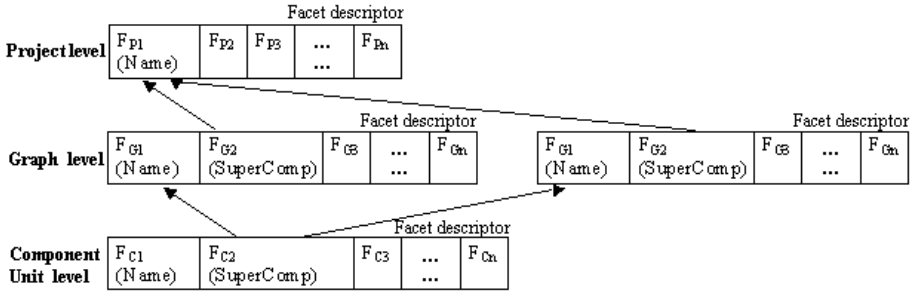

Fig. 4. The traceability in a hierarchical concept description schema

Such a hierarchical concept description schema not only represents the component concept, but also avoids information redundancy. The facet descriptor provides a possible schema to specify the component query criterion. In the following, we attempt to define facets that specify the concept of components on the project level, the graph level, and the component unit level.

Project Level. A project level component is coarse-grained. Like a framework, it helps organise a way to deliver a methodology specification, or a service in a specific application domain. Normally, the component forms an assembly of (the graph level) components with associated semantic definitions. The latter binds them together. The component interface includes a unique name and a general specification of its functionality and semantics. Facets of a project are defined as a set:

{Name, Application domain, Methodology, NoOfGraph}

E.g. {UML, method engineering, GOPRR, 9}
{Phone call design, User interface, <UML, Methodology A>, _}

In the facet descriptor, the application domain illustrates the field(s) to which a project belongs, and implies its services; the methodology represent the disciplines and semantics applied in a project; and the NoOfGraphs indicates the scale and complexity of a project. Generally, the more graphs involved in one project, the more

complicated the project will be. Following the facet descriptor, there are two examples: one describes an implementation of a UML methodology by using GOPRR metamodelling language, and the other describes a *phone call design* project, which uses both UML methodology and in-house methodology "A" for system design.

Graph Level. A graph is smaller in scope than a project, but a complete specification of services, such as the specification of static objects and their relationships, the input-output transformations, and the activities concerned with time and state changes. These specifications form three basic aspects of information modelling ontology: structure, function, and dynamic behaviour [19-21], and thus form an important facet to describe the abstraction dimensions. Meanwhile, since system development goes through several stages which may be supported by different techniques, the graph components must distinguish between different abstraction levels, e.g. analysis, design, and implementation. The abstraction level and techniques can be taken as facets as well. The facets of a graph level component can be defined as a set:

{Name, SuperComp, Type, Abstraction aspect, Abstraction level, NoOfObject}

E.g. {State Diagram, UML, GOPRR, Behaviour, Design, 7}
{Phone call state diagram, Phone call design, State Diagram, Behaviour, Design, 11}

Each graph has a unique name. SuperComp forms a necessary facet to trace the integration relationship among the components that are in the same project, but on different granularity levels. Examples of *State Diagram* and *Phone call state diagram* are represented above. The *Phone call state diagram* is the model level instance component of the metamodel level component *Sate Diagram*.

Component Unit Level. A component unit is a definition of an object, a relationship, a role, or an instance of the meta data type. They are the primary data in the GOPRR specification. Their facets can thereby be defined using GOPRR properties, as show in the following set:

{Name, SuperComp, Type, Property, Behaviours}

E.g. {State(UML), State diagram, Object, <State name, Variables, Internal actions, Symbol transparent?>, _ }
{DialTone, Phone call state diagram, State(UML), _, play dial tone}

As can be seen, *State(UML)* is an object type component having properties such as *State name*, *Variables*, and so on. *DialTone* is an instance of type component *State(UML)*, which is a composition of *Phone call state diagram*, and have behaviour of *playing dial tone*.

Summary. In the last three sub-sections, we have defined the facets. For a detailed explanation on the facets and their values, please refer to the appendix. We should notice that it is impossible to predefine all facets and values for components, since the number and type of components is extensible as more projects are specified in MetaEdit+. Therefore, the facets and their values are not exhaustive, and can be extended at any time. Meanwhile, we should notice that a common problem with the facet-based approach lies in mishandling synonyms and misinterpreting words that have some lexical ambiguity [22]. To solve the same problem in component interface

representation in MetaEdit+, the facets can be specified using a limited vocabulary, and the specifications can be chosen from the controlled vocabulary list.

4.2 Content

The content is a substantial part of a component, e.g. a software component content includes an implementation of the functionality or service specified in its concept part. In MetaEdit+, the component content varies from one context to another. On the metamodel level, the component is a collection of GOPRR based concepts and syntax specified in a method. On the model level, a component can be represented in diagram, matrix, table, or source code.

4.3 Context

The context forms specific relationship types among components and the development environment in which the component is designed. It is more complicated than the concept and the content representation, but benefits largely component reuse. In [7], the context of a reusable software component is further defined as:

- Conceptual context – conceptual relationship between the component and other components,
- Operational context – the characteristics of the manipulated data, and
- Implementation context – implementation dependency relationship between the component and other components.

Each aspect of a component context in MetaEdit+ is a little different from a software component context. It can be represented as follows:

Conceptual Context. With the growing number of concepts and the increasing interdependency in an information system development methodology, models for system analysis and design become more and more complex. Conceptual dependencies between components exist. There are varied conceptual dependencies in different contexts, e.g. a definitional dependency in the component definition description, and the reuse dependency in the reuse context.

A *definitional dependency* from a concept CPx to a concept CPy is created if CPx is used in the definition of CPy [23]. In short, it represents integration relationships among components. MetaEdit+ uses Info Tools to represent such definitional dependencies between selected components or graphs (see Fig. 5). The left-side figure represents a definitional dependency of a metamodel level component *State*, which is a composition of the graph level component *State Diagram*. It includes property type component units, e.g. *State name*, and *internal action* for its definition. The right-side figure represents a definitional dependency of one instance of *State* on the model level, called *Dialling*. As shown in the list, *Dialling* is defined in Project *mcc* in three representations (a graph, a matrix, and a table) of the model *Phone call state diagram*.

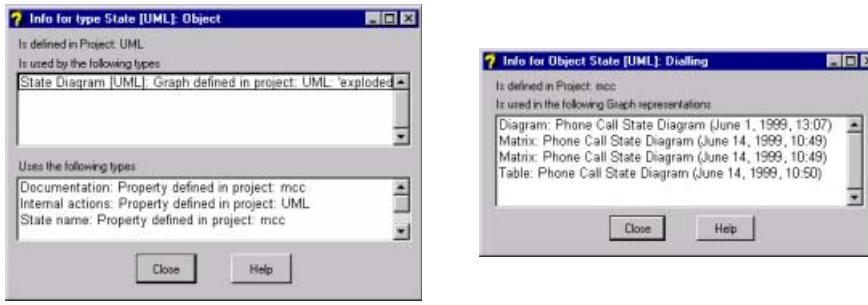

Fig. 5. Definitional dependency representation for the selected type/instance

Fig. 5 represents a definitional dependency of the graph level and the component unit level components. The same representation can be extended to project level components as well. It specifies a definitional relationship by relating a component's interdependent super- or sub-components. Such a relationship enhances users' understanding of a component on the whole, and thereby helps them select coherent sub-sets of a component and choose suitable ways (e.g. deep copy for components with significant changes or shallow copy for minor changes) for reuse.

When a component is built by reuse, *reuse dependency* specifies the reuse context between components. A reuse dependency relates a component with its original component(s) by specifying reuse associated information, such as the type of reuse, the way of reuse, modifications on the component concept and content, as well as the experiences collected in reuse processes. The contextual information effectively helps users study component features and outlines the possibility of reuse. For example, by inspecting the reuse dependency of an object component *State(UML)* in methodology UML, we learn that it is a shallow copy of the component *State* in OMT methodology. More properties such as *Internal actions* are added in *State(UML)*; and it is a horizontal reuse [24]. The reuse context implies that *State(UML)* refers to *State*, and any modifications on *State* will affect *State(UML)* as well.

Operational Context and Usage Context. An operational context defines characteristics of the manipulated data, such as types and operations available on the content of a component. It works on the code level component and organised to derive executable code. Since components in MetaEdit+ are mainly design artefacts, the operational context on source code limits the component information representation in MetaEdit+. We thereby define a usage context to specify the usage information among models.

Besides the concise functionality and the associated semantics abstracted in the component concept, a *usage context* records more detailed component usage information. It includes functional interdependencies among graph level components, suitable application domains and experiences accumulated, such as occurred problems and possible solutions. The usage context provides enriched information of using a component for user's reference.

Implementation Context. An implementation context describes how the component depends on other components for its implementation [9]. It provides a traceable relationship between components at the system analysis, design, and implementation stage, and helps users trace a component implementation in both forward and backward direction. We also call it a traceability feature.

In MetaEdit+, the metamodel level components are responsible for the methodology specification and construction. There is no implementation context between metamodel components. On the contrary, the model level components constitute the system development from the analysis stage to the implementation stage. Based on such a continuous process, we can define the component implementation context from its origins, through its development and specification, to its subsequent deployment and use, and through periods of on-going refinement and iteration in any of these phases. Such information provides a lifecycle of one component and helps users control the project as a whole, and finally improves the opportunity of reuse.

Summary. A component context is a complementary, but an important part that provides enriched contextual information including definitional dependency, reuse dependency, usage context and the implementation context. Without specifying the component context, the concept and reuse of components will remain ambiguous.

In MetaEdit+, Info Tools exist to represent definitional dependencies among components. Other contextual information may be represented in like manner. Moreover, we have hyperlink tools that represent the design rationale. The rationale provides basically understanding of why an artefact has been designed in a specific way, which may include information of e.g. requirements, assumptions, decisions, and alternative solutions [24]. Such hyperlink-based tools allow free hypertext linking of components. In the same way, we can define different types of contextual links to provide the access mechanism. By using the hyperlink functionality, component creators and users can easily to specify and navigate through the contextual information.

5 Conclusion

It is commonly accepted that components bring value to the information system development. In this paper, we have tried to extend component thinking to a metaCASE environment by defining components in MetaEdit+, a metaCASE environment. Accordingly, all types of design artefacts can be used as components after the necessary re-specialisation.

Component thinking brings several advantages to metaCASE tools. We identified three in particular. First, the definition of a hierarchical concept description makes components more comprehensible. The components have clearly defined interfaces based on predefined facet descriptors, which ease component search. Features of the project components can be shared across interfaces through a "SuperComp" facet, which avoids information loss and repetition. Second, the component context definition enhances component reuse. Different types of context represent conceptual relationships between the component and its environment, which makes it easy to

capture the complementary status information in a project, its dependence on other components, and its reusability. Such descriptions enhance component reusability in a metaCASE environment. Finally, component-based thinking enhances the usefulness of a metaCASE environment. MetaEdit+ has diverse facilities to describe the contexts among components, and support reuse activities. However, none of them are organised and applied in a systematic manner. Introduction of components lets us rethink existing facilities, and improve them systematically.

Although recent wide-scale emergence of the component concept and the accompanying CBD has fostered increased attention to components in industry as well as in academia, CBD in metaCASE environments are new. Besides potential benefits, several problems abound when introducing components. Although we defined the interface for each component, the distinction between interface and content is sometimes ambiguous on the unit level components. The content of a component unit may be as simple as features represented in its interface. Meanwhile, although we have already planned to implement the component framework in MetaEdit+, systematic technique support forms a challenge. Components are not limited to chunks of source code and have versatile forms and characteristics. Therefore, we still need more studies on how to apply different types of components in the same environment.

References

1. Slootman, F.: A blueprint for component-base applications. Compuware Corporation URL: http://www.compuware.com/products/uniface/station/reading/ind_blue.htm (1999)
2. Softera: SoftModeler. Softera Ltd URL: http://www.softera.com/products.htm (1997)
3. GmbH: MicroTOOL - making IT better. microTOOL GmbH URL: http://www.microtool.de /e_index.htm (1999).
4. Garone, S. and A. Kehlenbeck: Rational Rose98: Another Major Step Toward Component-Based Development. Rational Software Corporation URL: http://www.rational.com/ products/rose/reviews/analysts/index.jtmpl (1998)
5. Short, K.: Component based development and object modeling. Sterling Software (1997)
6. Kelly, S., K. Lyytinen, and M. Rossi MetaEdit+: a Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. Proceedings of the 8th International Conference CAISE'96. Springer-Verlag (1996) 1 - 21
7. Tracz, W.: Implementation working group summary. Reuse in Practice Workshop Summary (1990) 10 - 19
8. Harmsen, A.F.: Situational Method Engineering. University of Twente (1997) p. 310.
9. Whittle, B.: Models and languages for component description and reuse. ACM SIGSOFT: 20(2) (1995) 76 - 87
10. Edwards, S.H.: Towards a model of reusable software sybsystems. Proc. of the 5th Annual Workshop on Software Reuse (1992)
11. Brinkkemper, S.: Method engineering: engineering of information systems development methods and tools. Information & Software Technology: 38(6) (1996) 275--280
12. Smolander, K.: GOPRR: a proposal for a meta level model. University of Jyväskylä: Finland (1993)
13. Lyytinen, K., *et al.*: MetaPHOR: Metamodelling, Principles, Hypertext, Objects and Repositories. Technical Report TR-7. University of Jyväskylä: Finland (1994)
14. Rossi, M.: Advanced Computer Support for Method Engineering: Implementation of CAME Environment in MetaEdit+. University of Jyväskylä: Finland (1998)

15.Tolvanen, J.-P.: Incremental Method Engineering with Modeling Tools: Theoretical principles and Empirical Evidence. University of Jyväskylä: Finland (1998)
16.Prieto-Díaz, R. and P. Freeman: Classifying Software for Reusability. IEEE Software: (1) (1987) 6 - 16.
17.Prieto-Díaz, R. and G. Arango (eds.): Domain Analysis and Software Systems Modeling. IEEE Computer Society Press: Los Alamitos, CA (1991)
18.Liao, H.-C. and F.-J. Wang: Software reuse based on a large object-oriented library. ACM SIGSOFT: Software Engineering Notes: 18(1) (1993) 74 - 80
19.Iivari, J.: Levels of abstraction as a conceptual framework for an information system. Information System Concepts: An In-depth Analysis, P.l. E. D. Falkenberg (ed.). Amsterdam North-Holland. (1989) 323 - 352
20.Olle, T.W., *et al.:* Information System Methodologies -- A Framework for Understanding. Addison-Wesley (1991)
21.Wand, Y.: Ontology as a Foundation for meta-modelling and Method Engineering. Information and Software Technology: 38(4) (1996) 281 - 287.
22.Isakowitz, T. and R.J. Kauffman: Supporting Search for Reusable Software Objects. IEEE Transactions on Software engineering: 22(6) (1996) 407 - 423
23.Castellani, X.: Overviews of Models Defined with Charts of Concepts. IFIP WG8.1 International Conference on Information System Concepts: An Integrated Discipline Emerging (1999)
24.Oinas-Kukkonen, H.: Debate Browser - a Design Rationale Tool for MetaEdit+ Environment. Working Paper Series B 40. Univ. of Oulu: Finland (1996)

Appendix: Component Facet Descriptors on the Project Level, the Graph Level, and the Component Unit Level

Project level. The facet descriptor of a project is defined as:

{Name, Application domain, Methodology, NoOfGraph}
Where Name: Project name
Application Domain = {Method engineering, User interfaces, Communication and control system, Office management system, Real time system, Business application, … …}
Methodology = {GOPRR, OMT, OOD, OOAD, UML, … …}
NoOfGraph: the number of graph level components involved

Name provides a unique project name.

Application domain illustrates field(s) where a project operates and implies its services. If it is a metamodel level project to specialise and implement a methodology in MetaEdit+, its application domain belongs to method engineering. Otherwise, the model level project can be distinguished between the office management system, real time system, communication and control system, and so on. The value set of application domain can be augmented depending on projects involved.

Methodology represents the procedures, disciplines, rules and semantics that are applied in a project. A metamodel level project uses GOPRR metamodelling language as its methodology, while a model level project has more methodologies to choose. MetaEdit+ offers flexible method support, such as OMT (Rumbaugh et al.), OOD (Booch), OOAD (Coad/Yourdon), UML (Booch et al.), and the company's own methodologies that have been implemented in MetaEdit+.

NoOfGraphs presents the number of graph level components involved in a project. It indicates the scale and complexity of a project. Generally, the more graphs involved in one project, the more complicated the project will be.

Graph level. The facet descriptor of a graph is defined as:

{Name, SuperComp, Type, Abstraction aspect, Abstraction level, NoOfObject}
Where Name: Graph name
SuperComp: Collection of names of the related project level component
Type: The name of its metamodel (the name of the technique used)
Abstraction aspect = {Structure, Function, Behaviour}
Abstraction level = {Analysis, Design, Implementation, Maintenance}
NoOfObject: the number of objects involved

Name provides a unique graph name. On the metamodel level, a graph is a specification of a method/technique; on the model level, a graph describes one aspect within a specific application domain, e.g. a state diagram, a class diagram, and so on.

SuperComp illustrates a collection of projects in which the graph is involved. It forms a necessary facet to trace the contextual relationship among the components that are in the same project, but on different granularity levels.

Type defines similar information for the methodology facet on the project level. It presents the method/technique a graph used. On the metamodel level, the type for a method definition is GOPRR, while on the model level, there a different types of graph depending on the methodology used. For example, if a project applies methodology UML as the development methodology, the types of its graph can be Class Diagram, Use Case Diagram, Statechart Diagram, Activity Diagram, Sequence Diagram, Collaboration Diagram, Component Diagram and Deployment Diagram.

Abstraction aspect represents different scopes of information system modelling. Generally, these scopes can be distinguished between the structure which specifies static objects and their relationships in information modelling, the function which indicates data input-output transformation, and the dynamic behaviour which describes aspects concerned with time and state changes [19-21].

Abstraction level specifies the development stage a graph belongs to. On the metamodel level, it indicates the development stage this method/technique supports; on the model level, it represents the development stages that a diagram represents.

NoOfObject presents the number of objects involved in a graph. In the same manner as NoOfGraph, it indicates the scale and complexity of a graph.

Component unit level. A component unit facet descriptor is defined as:

{Name, SuperComp, Type, Property, Behaviours}
Where Name: Component unit name
SuperComp: collection of names of the related graph
Type = {Object, Property, Relationship, Role, type name}
Property: A collection of attributes of the component unit
Behaviour: A collection of operations included in the component unit

Name provides a unique name of a component unit.

SuperComp illustrates a collection of graphs in which the component unit is involved. It forms a necessary facet to trace the contextual relationship among the components that are in the same project, but on different granularity levels.

Type represents a metamodel level component unit as an object, a property, a relationship, or a role. The name of the metamodel level component forms the type of its instances on the model level. For example, *State* is an object type component unit in a State diagram specification. *DialTone* is an instance of object *State*, and its type is state.

Property represents a collection of attributes that a component unit has.

Behaviour represents a collection of operations included in a component unit.