Abstract

If we want to compose components which were used to build a certain software application, with other components to develop a new application, we lack the necessary knowledge to reuse these components. The research on software libraries has improved reuse. Our goal is to classify software components in libraries using a multi-dimensional approach supporting reuse as well as evolution. For this purpose, ontologies will be used capturing the structure of software component libraries, Description Logic will be used to build these ontologies and the component and composition patterns approach [VW01] [WV01] will be used to support reuse. This will provide the developer with an extended support to develop an application using components and with support to manage and maintain the software component libraries.

1 Introduction

Component-based software development (CBSD) is one of the major efforts for improving reusability and maintainability of software applications. A component was defined at ECOOP 96 [SP97] as follows: 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. The great advantage of component-based software development is that new software can be built by combining bought and self-made components. To do this, it is necessary to specify in which contexts the component can be used and how it will behave. This specification should enable the user of a component to determine whether it can be used in a particular case.

In [BJ99] a component specification consists of four levels, namely the syntactic or API level, the behavioural or semantic level, the synchronization level and the quality of service level. In relation to the previous, in our lab a component composition environment PacoSuite [VW01] [WV01] has been developed which supports the development of applications through the visual composition of software components. The components are documented with usage scenarios and component composition patterns. Both kinds of documentation make use of an extension of Message Sequence Charts (MSC). These diagrams describe typical role interactions of the components in a similar way as the interaction of objects is expressed in a UML interaction diagram. Component composition patterns are high level descriptions of cooperations between several roles without any indication on how this cooperation will be implemented. The same diagrams are used to
model typical interactions of a component with its environment, i.e. usage scenario’s. Based on that documentation automated compatibility checks are performed using finite automata. That research focuses on the synchronization level.

On the other hand, an important effort for improving reusability of software is the research done on software libraries. A software library is defined as a ”managed collection of software assets where assets can be stored, retrieved and browsed” [AM]. An important improvement in software libraries is the use of the faceted classification approach. A facet is a ”clearly defined, mutually exclusive, and collectively exhaustive aspects, properties or characteristics of a class or specific subject” [T92]. Each facet consists of several terms which describe the concrete values the facet can have.

In this paper we will bring together the aforementioned research on CBSD and the research on software libraries. We want to classify components (in the sense of CBSD) in software libraries by using the faceted approach. We will mainly focus on the semantic level by using ontologies. These ontologies will represent a multidimensional classification of components and will capture the behaviour of components. We envision this research as a way of improving reuse and evolution of component-based applications.

In section 2 the approach we follow will be explained. In section 3 we focus on the functionality and reuse dimension which are part of the ontology for generic components that we have developed. In section 4 we show how some components of a small application are classified using the presented approach. Finally, in section 5 we conclude and give some future work.

2 Our approach

We observe that software libraries using a faceted approach have some limitations:

- The description of the behaviour of the software artifacts, that are to be stored in the libraries, is possible if and only if they are fine-grained. However, we can have multi-behaviour components offering several services.

- Another complication appears whenever we try to classify components having state. In this case, the behaviour of a component can depend on the state the component has, which gives multiple ways of classifying the same component.

- If two software libraries of related domains are to be merged, currently, the only way of doing it is to manually add components one by one from one library to the other. This implies that the user must give the specifications for every component, which is a long and error-prone process. The best case is when in both libraries the same terminology is used, i.e. the same facets and terms are used. Otherwise, the user must also interpret each of those facets and terms and map them into concepts of the other library.

It is our intention to classify components in software libraries by using the faceted approach [PD91] [PF87]. This allows to have a multidimensional classification of components. We define a dimension as a set of facets that are related to the same view or the same aspect of a component. Different dimensions should be considered in the classification. For instance the functionality (what it does, inputs, outputs, etc.) of a component, the knowledge of its past uses (in which systems it has been used, with which components it has collaborated, with which composition patterns it has been used, etc.), the implementation issues (programming language, platform, etc.) and so on.

Furthermore, next to the different dimensions, also the relations among those dimensions (i.e. interdimensional relations) and the relations within one dimension (i.e. intradimensional relations) should be considered. As an example of
an interdimensional relation consider a network component sending packages to another component. Depending on the platform the performance of the component could be different. As an example of an intradimensional relation consider a component whose platform must have another value depending on the programming language (e.g. in Java, components should work on any platform). Our goal is to improve reuse and evolution in the development process of component-based systems. However, in this paper we will focus mainly on the reuse aspect.

2.1 Application-Specific Ontologies

To overcome the difficulties encountered in software libraries, we will describe the different facets of a component and their possible values together with the relations that exist between them in the same or another dimension. In other words, we want to create a layer that defines the structure of the component library. Ontologies describing the structure of the multi-dimensional classification of components will be constructed using an ontology language based on a Description Logic (DL). These ontologies will define the facets, terms, dimensions, inter- and intra-dimensional relations, etc. of the different components. The way this ontologies are defined depend largely upon the kind of application that can be constructed with these components. When using a component, the developer acquires knowledge about its use. When building new applications with that component the developer wants to be able to apply this knowledge. Component libraries are built based on such ontologies. The advantages of having such ontologies are:

- The insertion of the components in the library becomes easier because of the explicit presence of the relations. Given a term of a facet, other terms can be automatically derived due to those relationships.
- Different libraries based on the same ontology can be easily merged.
- Different libraries based on different ontologies can also be merged. In this case, first the ontologies should be merged. Some research is already done in this area [NM99], however it is still very preliminary. After merging the ontologies, the libraries can be merged based on the new ontology.
- Using the ontology, smart queries can be executed on the different software libraries.

2.2 Description Logic

The family of Description Logics originate from knowledge representation research in Artificial Intelligence. Their main strength comes from the different reasoning mechanisms they offer. The complexity of reasoning in these different languages is and has been widely investigated.

The basic elements of a Description Logic are concepts and roles. A concept denotes a set of individuals, a role denotes a binary relation between individuals. Arbitrary concepts and roles are formed starting from a set of atomic concepts and atomic roles applying concept and role constructors.

An ontology language will be developed based on the Description Logic $\mathcal{Q} - SHIQ$ [CL02]. The advantages of using Description Logic to build these ontologies are:

- Concepts can be easily composed to form new concepts.
- DL allows for arbitrary binary relations which enables the expression of the different relations between the components, their terms, facets and the dimensions.
- DL offers efficient reasoning support. This support can be used to reason about and to query the constructed ontology.

We will use the OIL ontology language [FH00] for the examples shown in this paper. This is a
language for specifying and exchanging ontologies. It is based on the DL $\mathcal{SHIQ}$ [HS01], on frame-based systems and on the web standards XML and RDF.

3 Functionality & Reuse Dimension

An ontology defining the structure of a software library for generic components has been defined. For space restrictions, we focus only on the two most relevant dimensions for reuse: the functionality and the reuse dimensions. As we have said, a dimension is composed by a set of related facets. A facet in our case can have more than one term associated. The reason for this is mainly that components are intrinsically reusable entities, and in consequence, they can be used in different contexts for different purposes. Therefore they can have several functionalities, different contexts where they can be used, different provided interfaces, and so on. Thus in order to specify this multi-context behaviour using the faceted approach, it is necessary that facets have the possibility of being linked to many terms. The specification of these dimensions has been figured out by inspecting and analyzing component-based applications and their components. We have to mention that this specification is still evolving.

Two kinds of facets are present: the ones that are useful for classifying components, and the ones that are not useful for classifying but for documenting. An example of the first kind is the facets that belong to the Functionality dimension, and of the second kind the facet Protocol of the Reuse dimension. Although the second kind is not useful for classifying, it is indeed useful for reusing components in systems that are being created, maintained or in systems that evolve.

3.1 Functionality

Every facet that has to do with the functionality part of the component belongs to this dimension. The components in this dimension are to be classified by their behaviour. This behaviour can be multiple due to the fact that a component can perform several actions. Some facets of this dimension are:

1. Actions: the different functions the component can perform. Some terms of this facet are:
   - Add: append, prepend, insert
   - Remove: delete last, delete first, delete any
   - Link
     - Out: reference, subscribe, unsubscribe, connect, disconnect
     - In: referenced, subscribed, unsubscribed, connected, disconnected
   - Display
   - Data: send, receive, set, get, notify, stream, answer
   - Calculation: sort, search, perform_specific,...

2. Inputs: arguments needed to perform some action.
   - String
   - Number: integer, float, ...
   - None
   - ...

3. Outputs: results of the performed action.
   - String
   - Number: integer, float, ...
   - None
   - ...

4. Medium: entities that are locales where the action is performed.
   - Dictionary
5. Kind: Information about the kind or type of the component.
   - Gui: button, console, ...
   - Data Structure: stack, list, tree, ...
   - Algorithm: sorting, searching, ...
   - Network: client, server, ...
   - ...

A hierarchy of terms has been created which structures all the possible values that a facet can have. The hierarchy presented above has been thought for classifying general purpose components. For classifying components that are specific of a given domain, other facets and terms (described in another ontology) should be used. There are also some relations between the terms of this dimension, but they will be described in 3.3.

### 3.2 Reuse

In this dimension the information that is useful at the moment of reusing the component in a given application is stored. The knowledge of past experiences (past uses) of the component is stored. The following are the 4 facets of this dimension:

1. Environments: Names of composition patterns in which the component has been previously used.
3. Related components: components that have been frequently used together with the component to be reused.
4. Related Systems: systems in which the component has been previously used.

Notice that only the third and fourth facets are useful for classifying. Indeed, the components can be grouped together following the values of those facets. On the other hand, the first and second facets are just for documenting. In other words, it is not possible to classify components by their corresponding MSCs because they are just drawings describing the manners that the components can be used.

### 3.3 Relations

Some relations have been found while defining the dimensions. Some of them are intradimensional and some interdimensional. For instance, there is an implicit relation between two terms of the functionality dimension, namely Gui and Display. Indeed, if the Display term is set, then there must be set also a term Gui, or vice versa. The relation can be described in $\mathcal{SHIQ}$ as follows:

$$
\forall \text{has\_kind.Gui} \sqsubseteq \neg \forall \text{has\_actions.Display}
$$

The relations can be helpful for automating part of the process of classifying a component into a software library. The selection of a particular term in a facet can trigger the automatic selection of other terms in other facets either in the same dimension or other dimensions or both. The previous process enforces the selection of some terms by inferring them from the specified relations, thus, preventing the user for making useless effort.

Both inter- and intradimensional relations can act on two levels: on the level of the ontology and on the level of the component (that must be specified by the user). The former is part of the definition of the structure of the library and indicates the relations of facets and terms that any component of the domain has. The latter is the relations that are inherent to a particular component. For instance, a typical relation in this level is when a given action in the functionality dimension is
linked to some particular inputs and outputs. The example given above belongs to the first level. An example of the second level of relations will be given in the following section.

4 Example

In this section we present how some components of a small application have been classified using the presented approach. This application is a scrabble game which has two players (Master and Slave) that communicate by means of two Network components that are connected with each other. The scrabble user interface component contains the interface of the game together with the logic of the program, the spelling checker receives a word and checks whether it corresponds to a valid English word, the Network component communicates with another Network component, and the Java button displays a button on the screen while it is waiting for a click to send a signal to another component in order to perform a given action.

The mentioned application is composed of the following components and composition patterns (i.e. the way the components communicate with each other) on the Master side:

- Scrabble user interface (ScrabbleGUI)
- Network client (Network)
- Spelling checker (Dictionary)
- Game_Master: composition pattern

On the Client side, it has in addition:

- Standard Java button (JButton)
- Game_Slave: composition pattern

As it can be seen, there are three components that are reused in both the client and server side: ScrabbleGUI, Network and Dictionary. We describe only these 3 components because we focus on reuse.

1. ScrabbleGUI
   - Actions: display, perform_specific, subscribed
   - Inputs: none
   - Outputs: none
   - Medium: none

2. Network
   - Actions: connect, disconnect, receive, send
   - Inputs: none
   - Outputs: none
   - Medium: none

3. Dictionary
   - Actions: search, answer
   - Inputs: string
   - Outputs: string
   - Medium: dictionary

By looking at this example (in particular to the component Dictionary), the necessity of having a relation among the terms of Actions, Inputs, Outputs and Medium arises. In this case, the relationship must state that the action search looks up a string (the input) in a dictionary and returns another string (the output). This is a relation at the level of the component as mentioned in 3.3. In figure 1 the expression of the relation is shown in an OIL editor. It is our aim to provide the user with a simple ontology language to enable him to write relations down in an easy way. This implies that the user does not have to know anything about DL.

As this ontology is defined only for generic components, then the action that is performed by a very specific component such as ScrabbleGUI is defined as ”perform_specific”. If a more fine-grained classification is needed, then another ontology (in this case it can be some kind of ”Game Ontology”) can be added to the software library.
The classification of a component looks as a very time-consuming process. Nevertheless we think that the previous is normal given the potential complexity of the component itself. What it is possible to do (and that is our intention) is to provide the developer with tool support for making both the classification of components and their (re)use as simple as possible, using the reasoning capabilities of DL.

5 Conclusions & Future Work

Our approach brings together several other works on software libraries, DL, ontologies and CBSD. One of our major goals is to support the software developer with a component software library for improving reuse and evolution. DL will be used for describing ontologies containing the structure of the software library (dimensions, facets, terms and relations). In this software library multi-purpose components can be classified, in contrast with other libraries that only classify atomic software artifacts (such as "input-output functions"). However, as the components to be classified can have several functionalities, its classification process becomes more difficult. By having relations, which link terms at the level of the ontology and component, the mentioned process can be semi-automated (thanks to the reasoning support offered by DL) for making it as simple as possible.

A scheme of an ontology for general-purpose components has been presented with an example of an application that follows the ideas in [WV01] and [VW01]. Also, it has been explained that if other kinds of components (some components that belong to a specific domain) are to be classified, a domain specific ontology, which specifies another dimension-facet-term-relation structure, can be used. This makes possible to have several domain-specific software libraries depending on the context in which the components they store are is used. Furthermore if there are two software libraries of domains that are very similar, they can be merged by merging their corresponding ontologies.

In the future we envision also the use of our approach for improving evolution of component-based applications. It is possible to classify and group "similar" components, i.e. components that can be interchanged in an application (when maintaining or evolving it) with a relatively low effort.

Our aim is also to provide tool support. A generic software library should be built structured only by the general purpose ontology presented above. Other more specific ontologies should be defined as well for creating the mentioned domain-specific software libraries just by plugging them in.

Another work to do is the support for components that have different functionality depending on their state. We have defined the ontology for generic components (a part has been presented in this paper) but the issue of the state of the components has not been addressed so far.
References


