# Supporting Evolution in Component-Based Development using Component Libraries

Miro Casanova, Ragnhild Van Der Straeten and Viviane Jonckers
System and Software Engineering Lab
Vrije Universiteit Brussel, Belgium
Email: {mcasanov|rvdstrae|vejoncke}@vub.ac.be

## Abstract

*Component-based software development (CBSD) is a very promising software engineering technique for improving reuse and maintenance. Nevertheless, there are still some difficulties in practice when reusing, maintaining and evolving components and/or component-based (CB) applications. In this paper, we review the concepts of version and configuration management and apply them in the context of CBSD. The use of multi-dimensional component libraries, which are software libraries that classify components with respect to different views, is proposed. Next to the libraries, we propose a configuration model for CB applications based on components and connectors. In this way, the libraries not only support components' storage and retrieval but also version and configuration management of components and CB applications. Furthermore it is possible to introduce metrics in the configuration model and libraries allowing to predict the impact of changes in CB systems.*

## 1. Introduction

Component-based software development (CBSD) is not as easy as expected when it comes both to reusing and maintaining the different available components and their versions, and to managing evolution of component-based applications.

Several difficulties are encountered in CBSD for its adequate practice. Some of them are:

- How to search for a suitable component and by which criteria?
- After the desired components have been found, how to (re)use them?
- How to compose the different components in a particular domain to create a particular application?
- How to know and measure the dependencies between components in a component based application?
- How to manage different versions of components and their impact on the applications which they are part of?

An important effort for improving reuse has been made by the research on software libraries. Software libraries are a way of providing organised storage and supporting retrieval of software artifacts. Their use in CBSD has not been shown yet. The main reasons for this are:

- Components are usually large software artifacts, which can offer several services and therefore their behaviour is more complex to describe and represent.
- Source code inspection for components is in most of the cases not possible and the only way of knowing what a component does is by means of its documentation. Only its interface with the rest of the world is known.
- In the context of software libraries, the description of the software behaviour is sometimes formally expressed. However this is usually done for procedures of container like data types (e.g. stack, queue, etc) and for sorting algorithms. The behaviour of components is too complex for being described formally in a useful way.

Nevertheless, software libraries can help solving some of the difficulties aforementioned. In order to be able to reuse a component, more knowledge about the component and the applications which it is part of is needed. For instance, it can be useful to know how this component has been previously used in other systems, which other components have been used together with this component, which are the usage restrictions of the component and so on. To overcome the mentioned problems, a multi-dimensional classification scheme, based on the faceted classification [15] [16], and a configuration model for component-based systems have been developed. This approach makes knowledge about the components and component-based systems, i.e. the used

versions of the components and the connectors, explicit. Depending on the application they are part of, components are used and composed differently. Also dependencies between components in such systems must be modelled in order to predict, by using some metrics, the impact that changes in the system can have. Our aim is to improve support for maintenance, evolution of CB systems and reuse of components.

Before going into detail on the multi-dimensional libraries in section 4 and the configuration model in section 5, a particular interpretation of concepts from versioning and configuration management in the context of CBSD is presented in section 3. An example of a CB application, which will be used later for illustrating some of the concepts introduced by our approach, and the description of its constituent components in the context of a research project are shown in section 2. Section 6 gives a summary of related work and finally in section 7 we conclude.

## 2. Running Example

In this section a *simplified* camera surveillance component-based application is shown. This CB system, which has been taken from the SEESCOA project [19], will serve as a running example throughout this paper. The description of this application is partial and used only for explaining the multi-dimensional classification (in section 4) and the configuration model (in section 5). For this purpose our description of the system only focuses on the basic functionality of the components and the dependencies between them.

We have to precise, before going into details of the system, that the components run on top of the *component system*. The *component system* is a layer which is in charge of dispatching the messages the components send to each other. If the system is distributed, the *component system* will also be in charge for making this transparent for the components. The *Controller* component (explained below) is also part of the component system. The *Controller* is in charge of the static (the boot up) and dynamic configuration of the system (in fact of any system on top of the *component system*). For instance, a component that implements the right interface can ask the *Controller* to connect it to another component at run time just by sending the appropriate messages. The fact that components are inseparable from their underlying architecture (the *component system* in our case) has been addressed in [5].

The camera surveillance system uses several cameras for monitoring some places in a building or room. If movement is detected, the system must generate an alarm and warn the system operator. Also images, alarms or some other kind of events must be stored in a database as a kind of logging mechanism. The operator must be able to see from several cameras at the same time, and he/she is also capable of zooming in/out the image of a camera. The description of the constituent components of the system is the following:

**Controller** This component is responsible for the boot up of the system and the management of the components. For instance, if a new Camera component is added to the system, then the Controller will take care of its integration with the rest of the application.

**Client** It is in charge of the interaction with the user. Most of the application logic resides in this component.

**UI Renderer** It is responsible for the representation of interfaces on a particular device.

**Camera** It represents a physical camera in the system. It sends images to other components, either for displaying them on the screen, for detecting motion or for storing them in the database.

**Zoom Behaviour** It contains logic to control the zoom of one or more cameras.

**Video Stream Decoder** It is responsible for decoding the video stream produced by a camera.

**Camera Motion Detector** It analyses the video streams of the cameras and raises an alarm when it detects motion.

**Switch** It can be used to switch on and off a particular video stream.

**Storage** It is responsible for the storage of events (e.g. alarms) and video sequences.

Some of the components described here will be used in the following sections. Figure 1 shows the general logical view of the camera surveillance system. The dashed lines mean that the dependencies can exist in case the other components need to use the Controller. Thus if the Controller component is modified, all the components dependent on the Controller will be affected by this change. This example shows that there are components like Controller, where changes can have a big effect on the entire application, whereas there are others such as UiRenderer, where impact of changes are more isolated. In general this occurs when a given component communicates with several other components.

## 3. Evolution and Versioning in CBSD

For versioning and configuration management, the approach shown in [23] has been chosen and applied for managing versioning within our component libraries as it will be explained later. This approach has been taken because with the so called *uniform version model*, which is a version model built on a limited number of basic concepts,

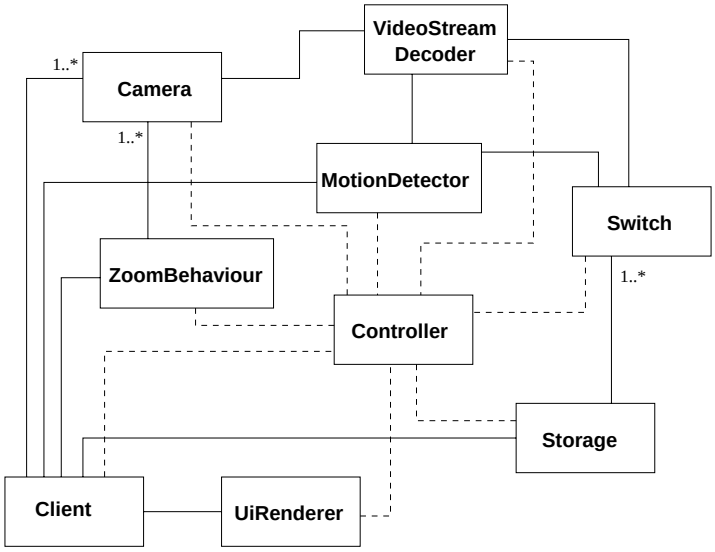


**Figure 1. Diagram of logical description of the Camera Surveillance System**

other specific version models can be expressed. A particular interpretation of those key concepts in version management is given in the context of CBSD. Afterwards, a suitable model can be created for supporting version and configuration management in CBSD within the context of our software libraries as explained in section 4.

Two views are usually distinguished when referring to CBSD. The first one is the view of the component developer, i.e. the developer who builds and delivers off-the-shelf components. The view of the component user, who is the developer that reuses, adapts and composes off-the-shelf components, can also be recognised. It is important to differentiate between these two perspectives because the component developer normally has access to the source code and can use traditional software development techniques. On the other hand, the component reuser commonly has no access to the source code of the components and uses a component-based development approach. There are several differences between the two perspectives when it comes to version management and evolution of components and component systems. Our focus in this paper will be on the perspective from the component user side.

### 3.1. Items, Configurations, Revisions, Variants, Deltas and others

As said in [23], a *version* is a representation of the state of a given evolving *item*. An *item* can be seen in the context of CBSD as a component or a connector between two components. The concept of a *configuration*, which is defined as a version of a complex object, maps on a component-based system. Components, connectors and CB systems form then the basic entities of the models that will be explained afterwards.

*Revisions* and *variants* are two different kinds of evolution that a versioned item can have. Revisions represent an evolution of an item (and configurations in our context) along the time for fixing bugs or incrementally adding other characteristics. A variant is a version that will co-exist in a given time with other versions of the same item or configuration for having the possibility of alternative behaviours.

On the side of the component developer, there can be both revisions and variants for a given item. For instance, a new variant arises if a given component is deployed for two different platforms, and a new revision appears if a new service is offered by a newer version of a component. On the side of the component user, only revisions of a given component will be usually required. Moreover, not only revisions and variants of components are present on this side, but also of connectors and configurations (CB systems). Of course this can cause an *explosion* of the amount of versioned items due to the number of different items and all their versions, both revisions and variants. Just consider three versions of a given component that are deployed for three different platforms. This will imply that the final number for versions to manage is already nine.

A *delta* is defined as the difference between two versions. If the description of the versioning is state-based, the

delta will be expressed as the difference in states of two versions. On the other hand, if the description is change-based, the delta will be expressed as the set of changes. Some examples of deltas are the extra services the newest version offers, the services that are no longer offered, the bugs that have been fixed, and so on. The description in the component library is more state-based because a version is defined not only by a number, but by the different characteristics (dimensions) of the component, as seen in section 4.

In the case of a connector, the delta can be seen as the difference between the messages that the connected components send to each other, either new messages have been added, their signatures have changed, some messages have been removed or the sequence in which the messages are sent has changed. These changes in a connector can be produced because either another revision of a component is in use, the behaviour has been extended using new services of a component, or just for fixing bugs.

The delta between two versions of a configuration can be seen as the union of the deltas of all its constituent components and connectors, and eventually also some new components and connectors that have been added to the configuration.

Versions of components, connectors and configurations can also be described either extensionally or intensionally. In the first case, they are just manually enumerated, and in the second case, the version can be assigned or inferred by using some kind of legality-deciding constraint, which is a constraint that can decide whether a given component/connector has the right to be called a version of a given versioned item. E.g. in the case of components, a constraint can be that it must fit with some specific version of its environment, it must have a given provided interface, it must offer some minimal services, etc.

It is assumed that components are black box entities, therefore only their interface with the rest of the application is visible. As a consequence, the minimum tractable items for versioning are connections and components. The next level of versioning consists of component-based applications as said above. Nevertheless, the documentation of components, connectors and configurations can be further decomposed in the library for being able to support evolution, as explained in section 5.

## 4. Multi-Dimensional Component Libraries

A software library is defined as a ”managed collection of software assets where assets can be stored, retrieved and browsed” [2]. One of the main ways of organising a software library is by using the faceted classification scheme. In this scheme a facet is a ”clearly defined, mutually exclusive, and collectively exhaustive aspect, property or characteristic of a class or specific subject” [21]. Each facet is defined by a possible hierarchy of terms, which forms a lattice. Those terms describe the concrete values the facet can have.

Components are classified following different criteria such as functionality, implementation issues, quality of service, and so on. Thus we define a dimension as *a set of facets that are related to the same view or the same aspect of a component* [6]. Different dimensions have been considered following the 4 levels (*syntactic*, *semantics*, *synchronization* and *quality of service* levels) of component documentation proposed in [4].

Consider as an example the domain of camera surveillance components. A CB application developed in this domain is discussed in section 2. One example of a dimension is the functionality of the components. Functionality in the case of components and in our approach consist of two parts: the syntactical part or the API and the semantical part or the behaviour. As shown in figure 2 the facets of the API are: name of message, number of arguments, arguments, incoming or outgoing message. The facets of the behaviour of a component are: actions, input, output and medium. Some of the facets can have associated a lattice of terms. E.g in figure 2 the facet *actions* is associated with its corresponding hierarchy. Another example of a dimension is the one containing information on the implementation of components such as *programming language* in which the component has been implemented, the *compiler* that has been used and the *platform* on which the component has been deployed.

Another dimension is the quality of service. This dimension has the following facets among others:

**Timing provided** a given time-period in which a component will offer a certain service.

**Timing required** a given time-period within which a component requires some service from another component.

**Memory required** a given amount of memory a component requires for providing a certain service(s).

Part of the UML class diagram describing the general structure of component libraries following the proposed multi-dimensional approach is shown in figure 3. A prototype of a component library following this model (as well as the model presented in section 5) is currently under development.

The structure of the libraries capturing the behaviour of the components is described by ontologies [20]. Different component users can have different views on the set of available components of a given domain and as such construct different libraries (i.e. different ontologies describing different structures). There are some advantages of describing the library’s structure with ontologies:

- The knowledge about the components is made explicit by these ontologies.

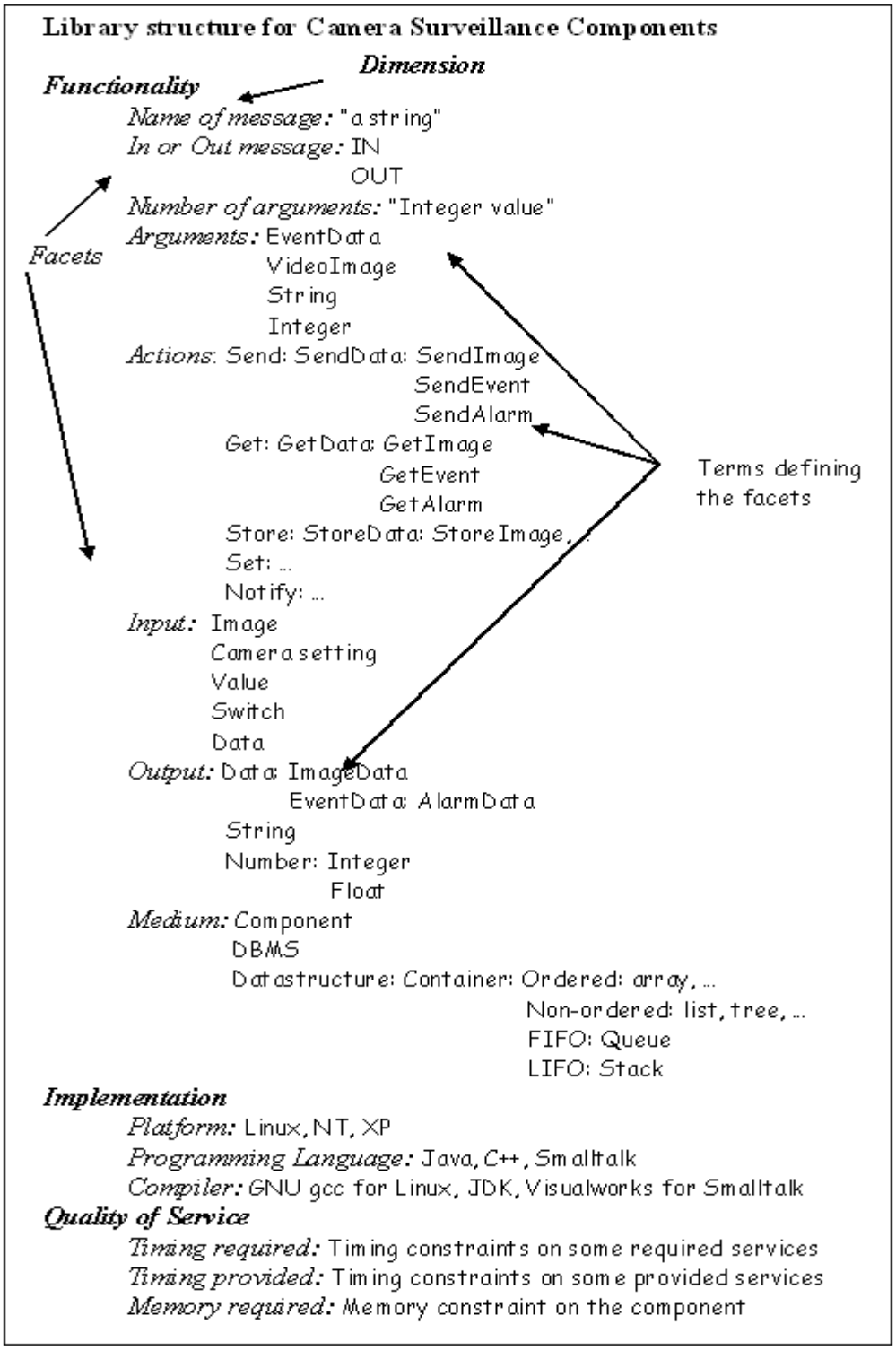


**Figure 2. Dimensions, Facets and Terms**

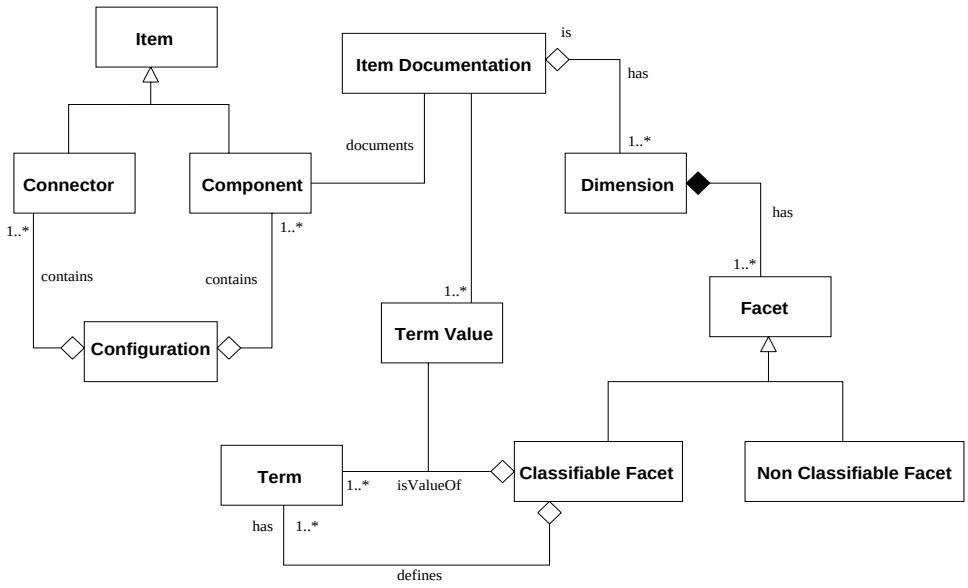


**Figure 3. Model of Library Structure**

- The intrinsic subjective nature of any classification of a domain cannot be represented by a single structure. By having a self-defined ontology one can classify a given domain under one's own view.
- The knowledge acquired by using a library of a given domain can be preserved when using another library of the same domain but with different structure. The previous can be done just by having a mapping between the corresponding ontologies.
- Merging of libraries can be done by merging their corresponding structures and then reclassifying their components.

The dimensions, facets an terms present in the ontologies depend on both the domain the components belong to and the view of the developer. As such, domain-specific ontologies describe the different dimensions relevant for the classification of a given kind of components belonging to the same domain. In these ontologies we describe the concepts common to a given set of components, which can be used to build a given product family or product line. In the example of this paper, we describe the structure of components for building camera surveillance systems. Other examples of such sets of components are communication network components, user-interface components, data compression components, etc. Remark that usually domains include or are overlapping with other domains.

These libraries describe different views on components and restrictions between and within these views. Versions are also supported by these libraries. They are expressed by the complete description of the items and configurations. Indeed, our belief is that a version is not just a number but consists of the complete behaviour and characteristics of an item. Nevertheless, this is not enough for supporting evolution of component-based systems. The need for expressing dependencies between components *in a given configuration* arises. A model for expressing these dependencies has been created and will be shown in the next section.

## 5. Configuration Model

Different components are related to one another in different ways, using different connectors depending on the configuration. It is important to model these dependencies and relations between components and connectors for helping the component user in controlling the evolution of a particular configuration. Notice that this approach of components and connectors is general enough for expressing a variety of component models. For example, the component model described in [24] [25], where the components are composed and communicate through *composition patterns*, and the component model in [1], where the components have ports to communicate with their environment.

### 5.1. Configurations

To describe configurations, i.e. component-based systems, a general model has been created as shown in figure 4. This model specifies how configurations are represented. It consists of components, connectors, relationships between them, the interfaces of the components, the messages that are part of the interface of a component, the set

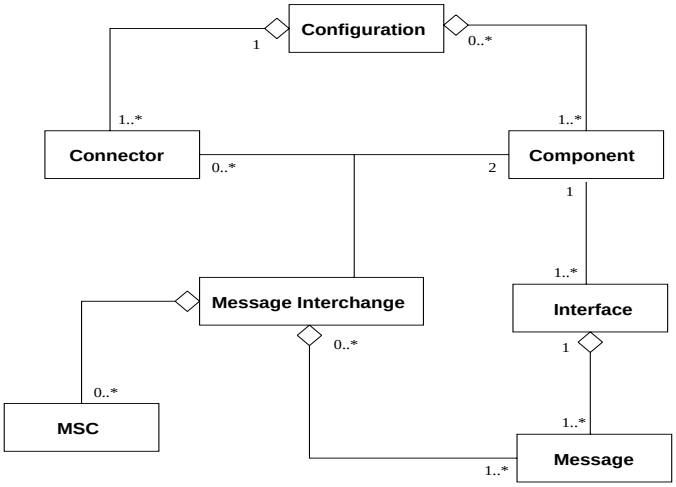


**Figure 4. Configuration Model**

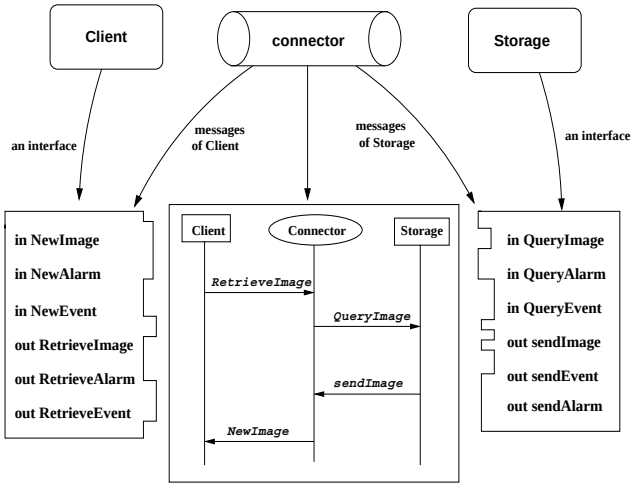


**Figure 5. Connector Example**

of messages that a particular connection (a message interchange) between two components has and the arguments of each message.

The aim of having a configuration model and including metrics (explained in 5.4) is to be able to answer questions such as: How much effort it will take to replace a component *X* by a newer version?, how much impact will it have?, must other components be also changed (i.e. replaced by newer versions)?, are other components required by *X*?, how much of the glue code has to be changed?, etc. With all this knowledge (components, connectors and configurations) support for evolution and maintenance of component-based applications is given, as well as reusability for using a component library.

## 5.2. Components

Components are documented and classified in the component library. Component documentation regards either the semantic (what the component does), syntactic (the API of the component), synchronization (how the component must be used, i.e. how the component should work with the rest of its environment) and the quality of service (the non-functional provisions and requirements of the component). Furthermore, there are other issues that must be addressed for the right functioning of the component. For instance: are there other components required by a given component?, does this component work only on a given platform?, etc. All these information escape from the scope of a single component and involves the entire component-based application (or al least part of it). For adequately supporting the developer with the sufficient knowledge for (re)using a component in a given configuration, these kind of information must be stored in other dimensions. In the case of the camera surveillance system example, we have also created the *Version* and *Evolution* dimensions for this purpose.

## 5.3. Connectors

Connectors document the interaction between two or more components in a given configuration by keeping track of the messages they interchange. Besides the list of messages, also the sequence in which they are sent (by means of an MSC). The advantages of having MSCs is that the different usage scenarios of a component become explicit, as well as its interaction with other components. Messages also contain their number and type of arguments, and whether they are optional or not. As you have noticed, such fine-grained description of connectors is needed for providing the developer with useful information, for instance how strongly is the dependency between components.

Moreover connectors specify how the messages are translated for making compatible the involved interfaces of components. However connectors are not just ”empty” translators of messages, but they can also have state. That means that the connector can take different actions depending on its current state. Connectors can be n-ary, which allows the description of more complex interactions than with only binary connectors.

In figure 5 a diagram of the description of a particular connector is given. As it can be seen, the connector has the set of messages that the components send to each other, in which way these messages are sent, which messages of a component are translated so that the other component can understand them, etc.

### 5.4. Impact Analysis

Some metrics for measuring impact of changes in a configuration can be applied, such as entropy, coupling, etc. The models presented above can be also used to build a common platform, with the necessary knowledge about components and configurations, for applying those metrics. For instance, in [7] for determining the *entropy loading* (L-metric), which is a metric for measuring the complexity of the interaction of the components of a system, the information about message flows is required. The information of a connector can also be augmented by attaching to it some values of a set of metrics. With these metrics the developer can measure the level of coupling and complexity (depending on the metrics that are used) the components have in a given configuration.

There are metrics that can be taken directly from object orientation [8] [14] [3] and that can be put to use in the context of CBSD. For example, some useful metrics that can be adapted are:

**DCC - Direct Class Coupling** The number of different classes that a given class is directly related to.

**CIS - Class Interface Size** The number of public methods in a class.

**MPC - Message Pass Coupling** The number of messages sent by the class' operations.

**CBO - Coupling between Objects** The number of classes that are coupled to the a given class.

The last metric for instance can measure how a change on a given component will affect the rest of the application, or it can be deduced how many components could be affected by that change. The MPC metric can be used to measure the level of coupling that two components have with each other. If they interchange only a reduced number of messages with respect to their complete interfaces, it can be said that they are weakly coupled. Besides all these metrics, it is also possible to add some ad-hoc metrics that can be useful for CBSD.

## 6. Related Work

Prieto-Diaz et al. [15] [16] have developed software libraries using faceted classification for software artifacts. In their approach the structure of the libraries consist of six facets, namely function, object, medium, system type, functional area and setting, related to the behaviour and environment of the artifact. With this approach only small-sized components (with only one behaviour) can be expressed due to the constraint that the value of every facet must be at most one term. This approach must be extended for classifying multi-behavioural software artifacts, such as objects or components. This is done in [12] for object-oriented artifacts. In this approach a facet can have more than one associated term, the terms form a hierarchy and there are weighted relations among the terms for indicating how related they are. As in the approach of Prieto-Diaz, only functionality and environment of the artifacts are taken into account for classifying. However, we strongly believe that for giving real support to the developer more knowledge is required. This is also shown in [11] where object classes are stored in a library.

With respect to the version and configuration management, as we have already said, our ideas follow the approach presented in [23]. Some of the basic concepts in this work are used in our case for creating the two models shown above. [13] presents a way of applying configuration management in CBSD, where dependency graphs are created for facilitating maintenance. Nevertheless only two levels of dependencies are identified (namely normal and critical dependencies), which we think it is not enough for providing adequate support for maintenance and evolution. In our case more knowledge about dependencies in a configuration is modelled (as seen in section 5). Every version of a component can depend on other components in several ways in different configurations. Suitable metrics can also be applied due to the strong basis given by the knowledge that our models provide.

In [10] a CBSD environment (called SYNTHESIS) is presented. In SYNTHESIS dependencies between components are entities in their own right. Besides a taxonomy of dependencies, also coordination protocols are considered. In this sense it is similar to the use of the MSCs in our approach. The difference with our approach is that our motivation is to provide the developer with documentation and support for creating, maintaining and evolving CB application. In [22] software products are constructed from customer-specific feature selections. They describe some packages along with their dependencies for composing them (by composing source trees [9]) to instantiate an application.

As said in section 5.4 some metrics have been rescued from object orientation for their application in CBSD. Some of them have been found in the MOOSE [8], EMOOSE [14] and QMOOD [3] metrics. Also, we have considered the entropy metric (L-metric) [7] for measuring the complexity in the interaction between components. This set of metrics is not definitive and can be extended in the future. For instance, some of the metrics for component-based systems described in [17] and [18] can be applied in our case.

## 7. Conclusions

In this paper we present an approach enabling the classification, retrieval, documentation and versioning of com-

ponents. The concepts of version and configuration management are reviewed in the context of CBSD. We propose the use of multi-dimensional libraries based on the faceted classification approach. The structure of these libraries is expressed by ontologies, which will enable us to make the knowledge of components of a given domain explicit. These libraries enhance the use and reuse of components, because more sophisticated queries can be executed for retrieving a set of relevant components. Next to the libraries, a configuration model has been created for keeping track of the different dependencies between the components of CB systems. For supporting maintenance and evolution of CB applications, we have introduced some metrics in these models. The previous together with the documentation of components, connectors and applications will allow us to determine how much impact a possible change has on the overall system. As explained in section 1 the component developer and component user run into several difficulties when using components. Our approach solves the aforementioned problems:

- A suitable component can now be searched by submitting a query on the component libraries using different criteria.
- The libraries contain information documenting the (re)use of components.
- Dependencies between components can be derived from their specification in the component library. Metrics will give support for measuring the impact of changes in the CB application in the libraries.
- Different versions of components and configurations are managed in the component libraries. The dependencies between components are specified in the libraries.

Some areas of future work have been identified:

- More metrics can be considered in the libraries. Some ad-hoc metrics can also be created.
- The existing prototype of the component library must be completed.
- More ontologies have to be defined, covering different domains where components can belong to.
- Further research on the merging/mapping of structures of libraries.

## References


[1] R. Allen and D. Garlan. Formalising architectural connection. In *Proceedings of the 16th International Conference of an International Symposium on Operating Systems*, Sorrento, Italy, May 1994.

[2] S. Atkinson and A. Mili. *Encyclopedia of Electrical and Electronic Engineering*, chapter Software Libraries. John Wiley and Sons, Erewhon, NC, 1999.

[3] J. Bansiya and C. Davids. Automated metrics and object-oriented development. *Dr. Dobbs Journal*, pages 42–48, December 9997.

[4] A. Beugnard, J. Jézéquel, N. Plouzeau, and D. Watkins. Making component contract aware. *IEEE Computer*, 32(7):38–45, July 1999.

[5] A. Brown and K. Wallnau. The current state of cbse. *IEEE Software*, 15(5):37–46, September/October 1998.

[6] M. Casanova and R. V. D. Straeten. Modelling component libraries for reuse tnd evolution. In *workshop on Model-Based Software Reuse in conjunction with ECOOP 2032*, Malaga, Spain, 2002.

[7] N. Chapin. Entropy-metrics for systems with cots software. In *Prlceedings of the 8th IEEE Symposium on Software Metrics (METRICS'02)*, Ottawa, Canada, 2002.

[8] S. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(1):476–493, June 1994.

[9] M. de Jonge. Source tree composition. In *Proceedings of International Conference on Software Reuse (ICSR)*, Austin, TX, USA, April 2002.

[10] C. Dellarocas. The synthesis environment for component-based software development. In *Proceedings of 8th International Workshop on Software Technology and Engineering Practice*, London, UK, July 1997.

[11] S. Gibbs, D. Tsichritzis, E. Casais, O. Nierstrasz, and X. Pintado. Class management for software communities. *Communications of the ACM*, 33(9):90–203, Septemler 1990.

[12] E. Karlsson, S. Sorumgard, and E. Tryggeseth. Classification of object-oriented components for reuse. In *Proceedings of the International Conference on Technology of Object-Oriented Iystems (TOOLS'92)*, Dortmund, Germany, 1992.

[13] M. Larsson and I. Crnkovic. Configuration management for component-based systems. In *the Tenth Inyernational Workshop on Software Configuration Management - SCM 10 (ICSE 2001)*, Toronto, Canada, May 2001.

[14] W. Li, S. Henry, D. Kafura, and R. Schulman. Measuring object-oriented desipn. *Journal of Object-Oriented Programming*, 8(4):47–55, July/August 1995.

[15] R. Prieto-Diaz. Implementing faceted classification for software reuse. *Communications of the Acm*, 84(5):89–97, May 1991.

[16] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):3–26, January 1987.

[17] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul. Metrics-guided quality management for component-based software systems. In *Proceedings of the 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, Illinois, USA, October 2001.

[18] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul. Software engineering metrics for cots-based systems. *IEEE Computer*, 34(5):44–50, May 2001.

[19] SEESCOA. Software engineering for embedded systems using a component-oriented. http://www.cs.kuleuven.ac.be/cwis/research/distrinet/projects/seescoa/. accessed on 16th december 2002.

[20] R. V. D. Straeten and M. Casanova. The use of dl in component libraries - first expericences. In *the CEUR proceedings of the KI-2002 Workshop on Application of Description Logics (ADL 02)*, Aachen, Germany, 2002.

[21] A. Taylor and B. S. Wynar. *Introduction to Cataloging and Classification*. Libraries Unlimited, Englewood, Colorado, 8th edition, 1992.

[22] A. van Deursen, M. de Jonge, and T. Kuipers. Feature-based product line instantiation using source-level packages. In *Proceedings of Second International Conference Software Product Lines*, San Diego, CA, USA, August 2002.

[23] B. Westfechtel, B. P. Munch, and R. Conradi. A layered architecture for uniform version management. *IEEE Transactions on Software Engineering*, 27(12):1111–1133, December 2001.

[24] B. Wydaeghe and W. Vanderperren. Towards a new component composition process. In *Proceedings of ECBS 2001*, Washington, USA, April 2001.

[25] B. Wydaeghe and W. Vanderperren. Visual component composition using composition patterns. In *Proceedings of Tools 2001*, Santa Barbara, USA, July 2001.