SOFA 2.0 metamodel

Petr Hnětynka, František Plášil, Tomáš Bureš, Vladimír Mencl, Lucia Kapová

Department of Software Engineering
Faculty of Mathematics and Physics, Charles University
Malostranské náměstí 25, Prague 1, 11800, Czech Republic
{hnetynka,plasil,bures,mencl,kapova}@nenya.ms.mff.cuni.cz

Abstract. In this report, we present a new version of the SOFA component model – SOFA 2.0. SOFA component model now seamlessly integrates a component-based technology with service-oriented technology. Such a technology merge takes advantages of both approaches and allows for better management of features like dynamic reconfiguration, supporting multiple communication styles, heterogeneous applications, etc.

1 Introduction

Component-based development (CBD) [23] has become a commonly used technique for building software systems. There are many opinions as to what a component is. One typically agrees that it is a black-box entity with well defined interfaces and behavior, which can be reused in different contexts and without knowledge of its internal structure (i.e., without modifying its internals). However, from a design view, components – especially hierarchical ones – can be viewed as gray-box entities with the internal structure visible as a set of communicating subcomponents. Typically, the collection of the related abstractions, their semantics and the rules for component composition (creation of component architecture) is referred to as a *component model* and an implementation of it as a *component system/platform*. In our view, the concept of "component" has always to be interpreted in the semantics of a particular component model.

Many component systems currently exist and are used both in industry and academia. Typically, the industrial component systems, such as EJB [7] and CCM [17], are based on a flat component model. On the contrary, the academic component systems and models usually provide advanced features like hierarchical architectures, behavior description, coexistence of component from different platforms, dynamically updatable components, support for complex communication styles, etc.

However, it is hard to properly balance the semantics of advanced features – in our view, this fact hinders a widespread, industrial usage of hierarchical component models. Based on our experience with the SOFA [20] and Fractal [4] component models, we claim that this issue primarily is related to dynamic reconfiguration of an architecture, i.e., adding and removing components at runtime, passing references to components, etc.

Another currently emerging paradigm is the service-oriented architecture (SOA) [25]. SOA-based systems (WebServices, etc.) are ordinary used in industry. In a high-level view, there is no difference between the SOA and CBD paradigms [11] – both a service and component have a well defined interface, their internal structure is not visible to their environment, and they can be reused in different contexts without modification. However, in SOA, services are not nested and their composition is typically made with the granularity of each request call, frequently being data driven. Thus, because of lack of any continuity in the architecture, there is no problem with dynamic reconfiguration.

In this paper, we use our experience with designing and implementing the SOFA component system [20]. SOFA employs a hierarchical component model and supports many of advanced features (like dynamic update, behavior description via behavior protocols, software connectors, etc.) and the prototype implementation is available as an open-source [22]. Although, it can seem that SOFA is an ideal environment for building large software systems, its usage brings several limitations. These limitations are – as it has been already said – mainly the problems with dynamic reconfiguration of applications and they are common for all hierarchical component systems. SOFA completely forbids the reconfigurations and allows just static architectures. Our solution of dynamic reconfiguration problem, which we describe in the paper, is in integration of ideas of component-based and service-oriented systems into a single seamless unit.

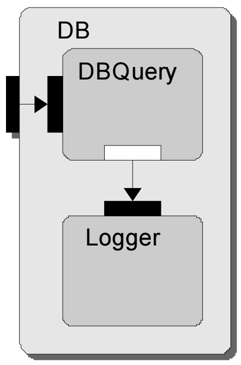
Other SOFA issues are problems with behavior specification and its collisions with different communication styles employed by connectors and bad extensibility of component controllers.

The goal of the paper is to describe a new version of the SOFA component model – SOFA 2.0 – which does not suffer of above mentioned problems.

2 SOFA Overview

SOFA is providing a platform for software components. It uses a typical component model with hierarchically nested components, which can be either primitive or composite. A composite component is built of other components, while a primitive one contains no subcomponents. A component is described by its frame and architecture. The frame is a black-box view of the component. It defines component's provided interfaces and required interfaces. The architecture is a gray-box view of a component; it implements component's frame by specifying subcomponents and their interconnections on the first level of nesting. Components are interconnected via bindings among interfaces. All bindings are performed using connectors [3], which are the first class concept like components. A behavior of SOFA components can be captured formally via behavior protocols [21].

A development lifecycle of a SOFA component is quite similar to other component systems. First, an ADL description has to be written. The description is then used to generate skeletons of the component implementation. A developer implements the components and inserts them into a repository. In order to launch the application, it is necessary to prepare a deployment plan, where components are assigned to concrete hosts and resources are allocated. Finally, according the plan, application is deployed and launched. Figure 1 shows an example composite component together with relevant parts of its ADL description.



```
frame DB {
  provides: IDBAccess db;
  protocol:
    (?db.open;?db.query*;
                    ?db.close)*
};

frame DBQuery {
  provides: IDBAccess db;
  requires: ILogger log;
  protocol:
    (?db.open{!log.log};
     ?db.query{!log.log}*;
     ?db.close{!log.log} )*
};
```

```
frame Logger {
  provides: ILogger log;
  protocol:
    ?log.log*
};

architecture DB implements DB {
  inst DBQuery db;
  inst Logger log;
  bind db:log to log:log;
  delegate db to db:db;
}
```

Fig. 1. SOFA application example

2.1 Dynamic reconfiguration

Based on our experience with the SOFA [20] and Fractal [4] component models, we claim that this issue primarily is related to dynamic reconfiguration of an architecture, i.e., adding and removing components at runtime, passing references to components, etc. A simple prohibition of dynamic reconfiguration (even though adopted by some systems [2]) would be very limiting, since dynamic changes of architecture are inherent to many component-based applications [15]. On the other hand, particularly in hierarchical component models, an arbitrary sequence of dynamic reconfiguration can lead to "uncontrolled" architectural modification, which is inherently error-prone (we call this issue the *evolution gap problem*). Moreover, for description of component architectures, most of the component models provide an architecture description language (ADL) [2,6,14,15], which typically captures just the initial components' configuration. (The idea of software architectures and ADL specification came from hardware design, which is static by nature). Thus a challenge is to somehow capture reconfiguration in an ADL.

However, in hierarchical architectures a key problem is where a newly created component and connection have to be added into the hierarchy and how to establish new connections – in particular the problem is pressing when the connection is not among siblings in the hierarchy.

Consider the situation on Figure 2a) capturing a fragment of a data accessing application, which logs all method calls to a set of loggers connected via a required collection interface. The DAccess is a data access component, which is bound to LFactory (the logger factory) and which features a collection requires interface for accessing the loggers. As a result of a call to its provided interface, the logger factory creates a new logger component and returns a reference pointing to it. Such a call is issued by the DAccess component, which thus receives a reference to a new logger and binds to it via the collection interface (dashed line on Figure 2a).

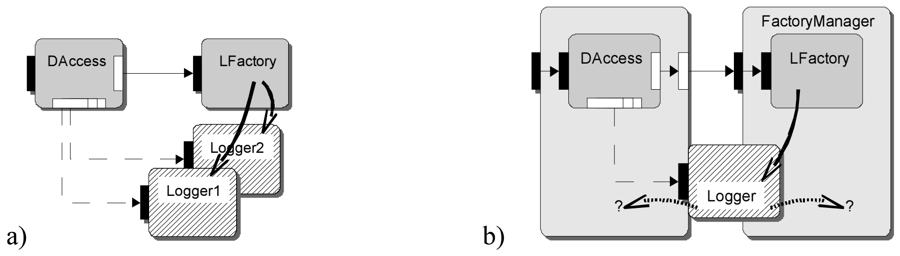


Fig. 2. Dynamic reconfiguration example

Provided the DAccess and LFactory components are siblings in the hierarchical architecture, there are no problems with such a dynamic reconfiguration. However, a problem arises when this is not the case as depicted on Figure 2b). The issue is, where the newly created component (Logger) should be placed in the architecture and how the connection to it should be established.

A straightforward answer to the question where to put dynamically created Logger components might be into the FactoryManager. However deciding how to manage their connections to DAccess is not that intuitively obvious.

If we allow a direct connection between the DAccess and Logger, then the connection will go through the FactoryManager component boundaries and violate the requirement of encapsulation. The second possibility, to add a copy of the Logger provided interface to the FactoryManager component and lead the connection through it is not also ideal, because it would mean that FactoryManager had to mediate traffic of all sessions. In general, if the component deciding on creation of another component is located in a different part of the hierarchical architecture than the new component is to be connected with, the problem of mediating connections becomes pressing.

2.2 Control part

In addition to "business" interfaces (i.e., normal interfaces with provided and required functionality of a component), components usually have so called *control* interfaces (*controllers*). From architectural view, these controllers are provided interfaces and they correspond to non-functional features of components, i.e., life-cycle management, reconfiguration, introspection, etc. The controllers should not be accessed by application logic but they are intended to be used by runtime environment.

Components in the current SOFA implementation provides just single fixed control interface called Component Manager (CM). CM provides all necessary non-functional features, which are management of component life-cycle (starting/stopping/updating component) and management of component interconnections.

As it has been said, CM is fixed for all components. But this approach brings several problems, because in many cases a component can suffice with smaller functionality (e.g., in environment with limited resources) or in other cases it would be desirable to have possibility of extending a set of non-functional features (like monitoring of incoming/outcoming calls, intercepting calls, etc.).

3 SOFA 2.0 Description

From the global view, the main concepts and general design of the SOFA system remains the same. The first-class entities are still components and connectors; SOFA still employs a hierarchical component model.

In the original SOFA system, components have been designed and composed using CDL description. This CDL language in fact defines the used component model. In the new SOFA system, we are leaving this ADL-based definition of the component model and we are using a meta-model based definition. More specifically, we are using the MOF technology [19] for designing the component model. The advantages of such approach can be found in [8].

3.1 SOFA component model

The relevant part of the meta-model of SOFA is depicted on figure in Appendix A. Using it, we describe the essential features of the SOFA component model. In the following text of the section, the terms in italics have direct representation in the meta-model.

The core element is the *frame*, which defines the black-box view of the component. It can provide and require a set of interfaces. Each *interface* has its *interface type*, which is defined its signature. The signature is a name of the type in an underlying language. In addition to the type, the interface has several other attributes. The *communication style* allows determining in which way components connected through this interface can communicate. The *communication feature* is tied with the communication style and allows further specification of features that the communication through the interface has to have. Then, the interface has specified its cardinality (single or collection) and contingency (optionally connected or mandatory). The last attribute, the *connection type*, which has two possible values – normal or utility, is tied with dynamic reconfiguration and is explained later.

On the other hand, the frame is implemented by an *architecture*, which represents a gray-box of a component. It is obvious, that a single frame can be implemented by several architectures. But also a single architecture can implement several frames. It is an analogy from object-oriented programming, where a single class can implement several interfaces.

The architecture of a primitive component is empty; the architecture of a composite component contains definition of *subcomponents* and *bindings* among these subcomponents. In original SOFA, the architecture of a composite component describes just one level of nesting – its subcomponents are defined using frames. In the new SOFA, the architecture can define more than a single level; subcomponents can refer to frames or directly to other architectures. In the meta-model, this fact is emphasized by comment (with *xor* label) between the *instantiatesFrame* and *instantiatesArchitecture* associations.

In addition to subcomponents, architectures can define *bindings* among these subcomponents. There are three kinds of bindings. A *delegation* connects a provided interface of component to a subcomponent's provided interface and a *subsumption* connects from a subcomponent's required interface to a required interface of component. The last kind is a *connection* between two or more subcomponents.

The *interface type*, *frame* and *architecture* elements have names and also version identification. The versioning model suitable for such distributed systems and which is already used in old SOFA, is described in [9].

The remaining elements are *properties* of frames and architectures, which are just name-type pair and they are intended for component parameterization at deployment time, and *annotations*, which allows to annotate frames and interfaces with additional information.

3.2 Dynamic reconfiguration

By *dynamic reconfiguration* we mean a run time modification of an application architecture. A special case is a dynamic update of a component supported by the original SOFA (and also in the SOFA 2.0); the principle is that a particular component is replaced with another one having a compatible interfaces. This kind of dynamic reconfiguration is easy to handle and there are no problems with the application's architecture, because all the changes are located in the updated component and are transparent to the rest of the application. Since the new component can have a completely different internal structure, such a component update in principle means replacing a whole subtree in the component hierarchy, being thus a "real" architecture reconfiguration. Also, as an aside, dynamic update is not usually initiated by the application itself but by an external entity (the user, provider, etc.).

A general dynamic reconfiguration is an arbitrary modification of an application architecture though. We have identified the following five elementary operations dynamic reconfiguration is based upon: (1) removing a component, (2) adding a component, (3) removing a connection, (4) adding a connection, (5) adding/removing a component's interface.

In hierarchical component models, as mentioned in Sect. 2.1, an arbitrary sequence of these operations can lead to "uncontrolled" architectural modification (the evolution gap problem). To avoid it in SOFA 2.0, we limit dynamic reconfigurations to those compliant with specific *reconfiguration patterns*. At present, we allow the following three reconfiguration patterns: (i) nested factory, (ii) component removal, and (iii) utility interface. In principle the operations (1) – (4) are to be employed in these patterns only, and the operation (5) is limited to the use of collection interfaces (an unlimited array of interfaces of a specific type in principle). The choice of these patterns is based on our experience gained out of non-trivial case studies.

3.2.1 Nested Factory Pattern

The nested factory pattern covers *adding a new component* and *a new connection* to an architecture. The new component is created by a *factory* component as a result of method invocation on this factory. The key related issues are (i) where in the hierarchy the new component should be placed, and (ii) how the connections of/to the new component should be lead.

Consider the situation on Fig. 2a) capturing a fragment of a data accessing application, which logs all method calls to a set of loggers connected via a required collection interface. The DAccess is a data access component, which is bound to LFactory (the logger factory) and which features a collection requires interface for accessing the loggers. As a result of a call to its provided interface, the logger factory creates a new logger component and returns a reference pointing to it. Such a call is issued by the DAccess component, which thus receives a reference to a new logger and binds to it via the collection interface (dashed line on Fig. 2a).

Provided the DAccess and LFactory components are siblings in the hierarchical architecture, there are no problems with such a dynamic reconfiguration. However, a problem arises when this is not the case as depicted on Fig. 2b). The issue is, where the newly created component (Logger) should be placed in the architecture and how the connection to it should be established.

A straightforward answer to the question where to put dynamically created Logger components might be into the FactoryManager. However deciding how to manage their connections to DAccess is not that intuitively obvious. If we allow a direct connection between the DAccess and Logger, then the connection will go through the FactoryManager component boundaries and violate the requirement of encapsulation. The second option, to add a copy of the Logger provided interface to the FactoryManager component and lead the connection through it is not also ideal, because it would mean that FactoryManager had to mediate traffic of all sessions. In general, if the component deciding on creation of another component is located in a different part of the hierarchical architecture than the new component is to be connected with, the problem of mediating connections becomes pressing.

In SOFA 2.0, we have adopted the following rule: The newly created component becomes a sibling of the component that initiated the creation (and its call to the factory also determines the component's collection interface the connection is to be established to). In the example above, the Logger component becomes a sibling of the DAccess component – see Fig. 3a).

The main reason, why the newly created component does not become e.g. a sibling of the factory component (as it can seem to be also an obvious simple solution) is that the component which initiated the creation typically needs to intensively collaborate with the new component which is obviously easier to manage with a sibling. The next positive outcome of the rule is better performance, because it is not necessary to create complicated connections going up and again down through the hierarchy (imagine DAccess communicating with Logger if it were a sibling of LFactory (Fig. 3b).

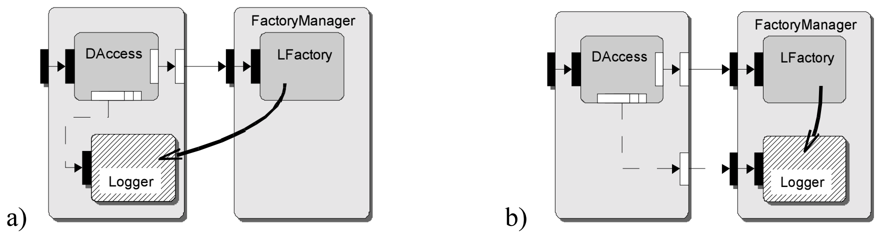


Fig. 3. Dynamic application example

Technically, to identify a factory component, *factory* annotation can be applied to factory methods of an interface.

The newly created component (NC) is not limited to having just a provided interface (as it is shown in Fig. 2 and Fig. 3) but it can have also required interfaces. However, these are restricted just to the types featured by the component initiating the creation (IC). At the moment the provided interface of NC is bound, the required interfaces are also bound to the same provisions as the required interfaces of IC are.

3.2.2 Removing component pattern

Removing a component can be seen as a quite common situation. The closely tied and necessary action is removing all connections to the removed component. An example of such situation can be again the data access component and set of loggers. A single logger component can be removed together with link from the data access component. From the view of the application architecture, there are no problems with removing components. The only issue is that after removing component and corresponding connection, the unbound required interfaces can remain. In SOFA, such unbound interfaces do not bring any problems because behavior of each component is described by behavior protocol and the component environment can verify, whether the required interfaces can be left unbound or not.

Removing connections is allowed just as the result of removing a component as it is described above.

3.2.3 Utility Interface Pattern

While working on case studies, we frequently faced the situation when a component provides a functionality, which is to be used by multiple ("almost all") components in the application (i.e. the need of use is orthogonal to the components' hierarchy). The functionality is typically some kind of a broadly-needed service such as printing. A solution can be to place such a component on the top level of the architecture hierarchy and arrange for connections through all the higher-level composite components to the nodes, where the functionality is actually needed. But this solution leads to an escalation of connections and makes the whole component architecture blurred (by making unimportant utility features too visible) and consequently error-prone. Another typical situation we faced is that a reference to such a service is to be passed among components (e.g., returning reference to a service from a call of a registry/naming/trading component).

For this reasons, we have introduced *utility* interfaces (see the meta-model in the Appendix). The reference to a *utility* interface can be freely passed among components and the connection made using this reference is established orthogonally to the architecture hierarchy (Fig. 4).

From a high-level view, the introduction of utility interfaces brings into component-based models a feature of service-oriented architectures. Such feature fusing allows to take advantages of both these methodologies (e.g., encapsulation and hierarchical components of a component model and simple dynamic reconfiguration from a service-oriented architecture).

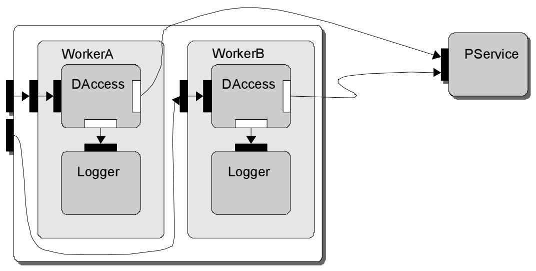


Fig. 4. Utility interface example

As a side effect, the introduction of utility interfaces this way consequently means that – in a limiting case – the whole application can be built only of components with utility interfaces and therefore the component-based application becomes an ordinary service-oriented application (inherently dynamically reconfigurable). Thus, service oriented architecture becomes a specific case of a hierarchical component model.

3.3 Control part of components

The control part of component is in SOFA v. 2.0 modular and is based on usage of aspects. The general idea of this approach and its application for Fractal component model is described in [16].

Using this approach, the control part of a component is modeled as a set of so called *microcomponents*. The microcomponent model is flat; a microcomponent cannot be composed of other microcomponents. Also to avoid recursion, the microcomponents do not have control part.

The control part of a SOFA component always contains several necessary microcomponents, which exposes interfaces as control interfaces of the component. These are the *lifecycle* controller, which allows for starting/stopping/updating a component, and *binding* controller, which allows for adding/removing connections among components. The other functionality of the control part can be added to a component as an aspect at deployment time.

The metamodel of aspects and microcomponents is shown in Appendix B. The meaning of elements in the metamodel is also described in [16].

4 Related Work

Related work: In the following paragraphs, we describe related work and we split it into two sections – work related to controllers and work related to dynamic reconfiguration of components.

The CORBA Component Model (CCM) [17] and Enterprise Java Beans (EJB) [7] are representatives of industrial component systems employing flat component model. CCM components' interfaces are classically divided into provides and required ones; in addition there is the next division into interfaces for synchronous and asynchronous invocations. Components are defined in IDL (Interface Description Language) but IDL does not provide any support for component composition. EJB components are defined directly in Java. Moreover, EJB components do not have explicitly specified required interfaces. Each EJB component has a *home* interface, through which new components can be created. The similar concept in CCM is called a *factory* interface. Both these interfaces can be seen as controllers. In addition, CCM components can have *attributes*, which are named values exposed via getter and setter methods and primarily intended for component configuration. These attributes can be again seen as a control interface. In both systems, these control interfaces are fixed and are not extensible in any way.

The Fractal component model [4] uses a classical component model with hierarchically nested components, In Fractal, each component can have multiple control interfaces. The number and types of control interfaces depend on configuration of a run-time environment. In the Fractal component model, each component is split to two parts – control part and content. The content is composed of other subcomponents or in the case of a primitive component it is directly implemented. The control part contains implementation of control interfaces and other elements implementing non-functional features.

Component systems with a flat component model (CCM [17], C2 [24]) do not consider dynamic reconfiguration as an issue, since only a flat architecture gets modified and a service can be seen as another component in the flat component space. However, the evolution gap problem is inherently present.

In the area of hierarchical component models, there are several approaches as to how to deal with dynamic reconfiguration.

(1) *Forbidding*. A very simple and straightforward approach used in several component systems (e.g., [2]) is to forbid dynamic reconfiguration at all. But this is very limiting, revealing in essence all the flaws of the static nature of an ADL.

(2) *Flattening.* Another solution is to use hierarchical architecture and composite components only at the design time and/or deployment time. However, at run time the application architecture is flattened and the composite components disappear – this way the evolution gap problem becomes even more pressing, since the missing composite components make it very hard to trace the dynamic changes with respect to the initial configuration. This approach is used, e.g., in the OMG Deployment & Configuration specification [18], which defines deployment models and processes for component-based systems (including CCM). The component model used in the OMG

D&C specification is hierarchical, but finally in the deployment plan, the application structure is flattened and the composite components are removed.

(3) *Restricted reconfiguration*. Several systems forbid an arbitrary reconfiguration but allow special and well defined types of dynamic reconfiguration,

(a) *Patterns*. ArchJava [1] is a component system employing a hierarchical component model. (Archjava is an extension of Java). Components in ArchJava can be freely added (using the *new* operator like for plain objects), but addition of new connections is restricted by *connection patterns*. These patterns define through which interfaces and which types of components the new component can be connected. Moreover, only the direct parent component can establish these connections (among direct sibling components).

(b) *Shared components*. Fractal introduces shared components (at ADL level); a shared component is a subcomponent of more than one other component. This way, component hierarchy becomes a DAG in general (not a tree). Appling this idea to the example in Fig. 1 would mean that the Logger component would be used by LFacory and DAccess. This solution works nicely, however, an architecture with shared components can be confusing, since it is not easy to determine who is responsible for lifecycle of a shared component, reasoning about architecture (e.g., checking behavior compliance) is very complicated and several advanced features of component models (e.g., dynamic update of a component subtree) cannot be applied.

(c) *Formal rules*. Several systems (e.g., CHAM [10], "graph rewriting" [27]) define a formal system for describing the permitted dynamic reconfiguration. These systems allow complex definition of all architecture states during an application's lifecycle. But they are very complicated, even for simple architectures.

(4) *Unlimited.* Darwin [14] uses direct dynamic instantiation, which allows defining architecture configurations that can dynamically evolve in an arbitrary way (but the new connections among components are not captured). Julia [12], an implementation to Fractal, allows a general component reference passing (so that any time a reference is passed it mimics establishing a new connection – this works orthogonally to specifying a shared component in ADL). Obviously, the evolution gap problem is ubiquitous in these cases.

On the other hand it should be emphasized that SOA is typically based on dynamic reconfiguration, since the composition of services is done with the granularity of individual calls captured in coordination languages like Linda [26] or routing of messages [5].

5 Conclusion

In this report, we present the new version of our component system SOFA. The features of the whole component model are defined using the meta-model. For handing dynamic reconfiguration, we allow several well defined patterns, which cover most common situations, where the reconfiguration is necessary. In order to allow nearly an arbitrary dynamic reconfiguration, we introduced so called utility interfaces, which in component-based system bring features of a service-oriented system. This integration helps to solve problems of hierarchical component models and allows for their wider usage.

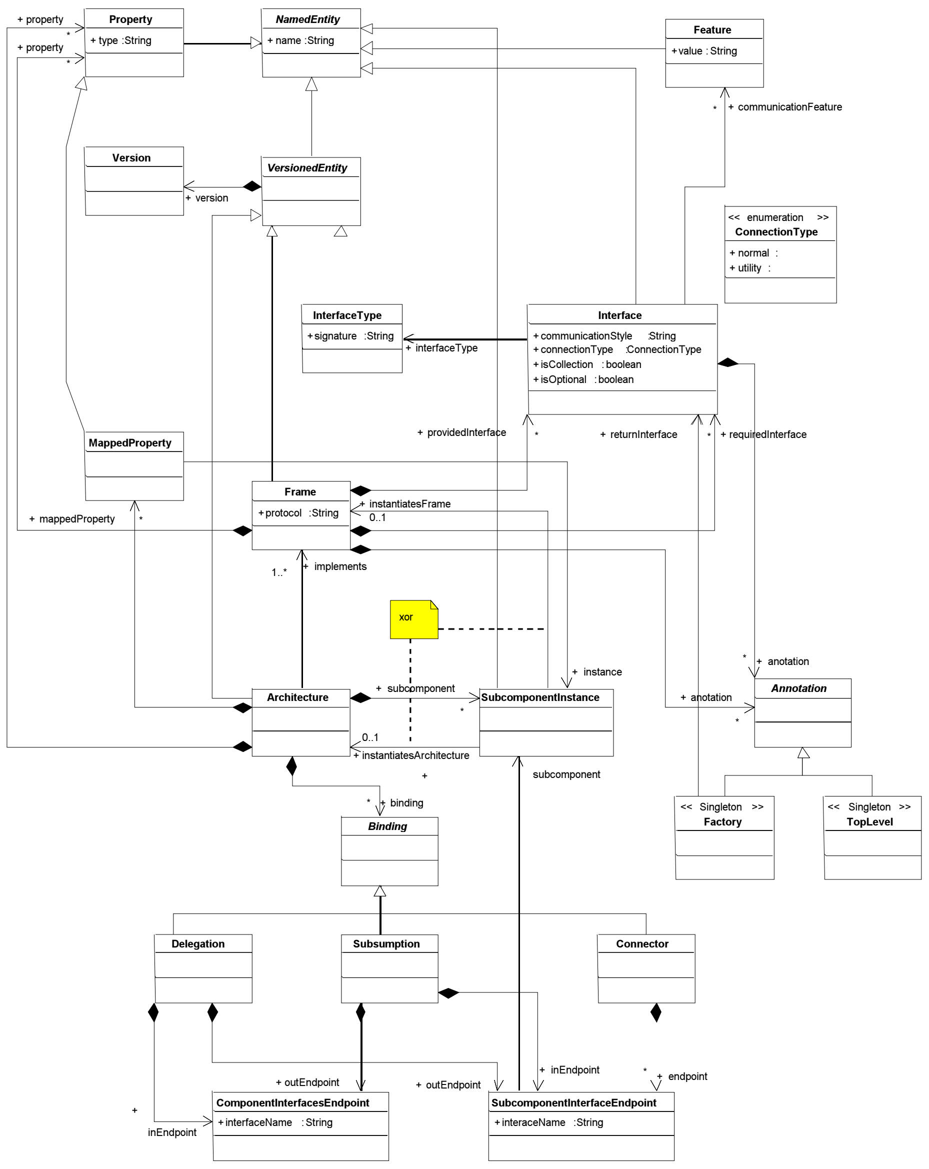
Currently, we have specified the whole meta-model of SOFA, all necessary interfaces for development time, deployment and runtime environments and we are implementing the whole system. The workable system is expected within several months.

References

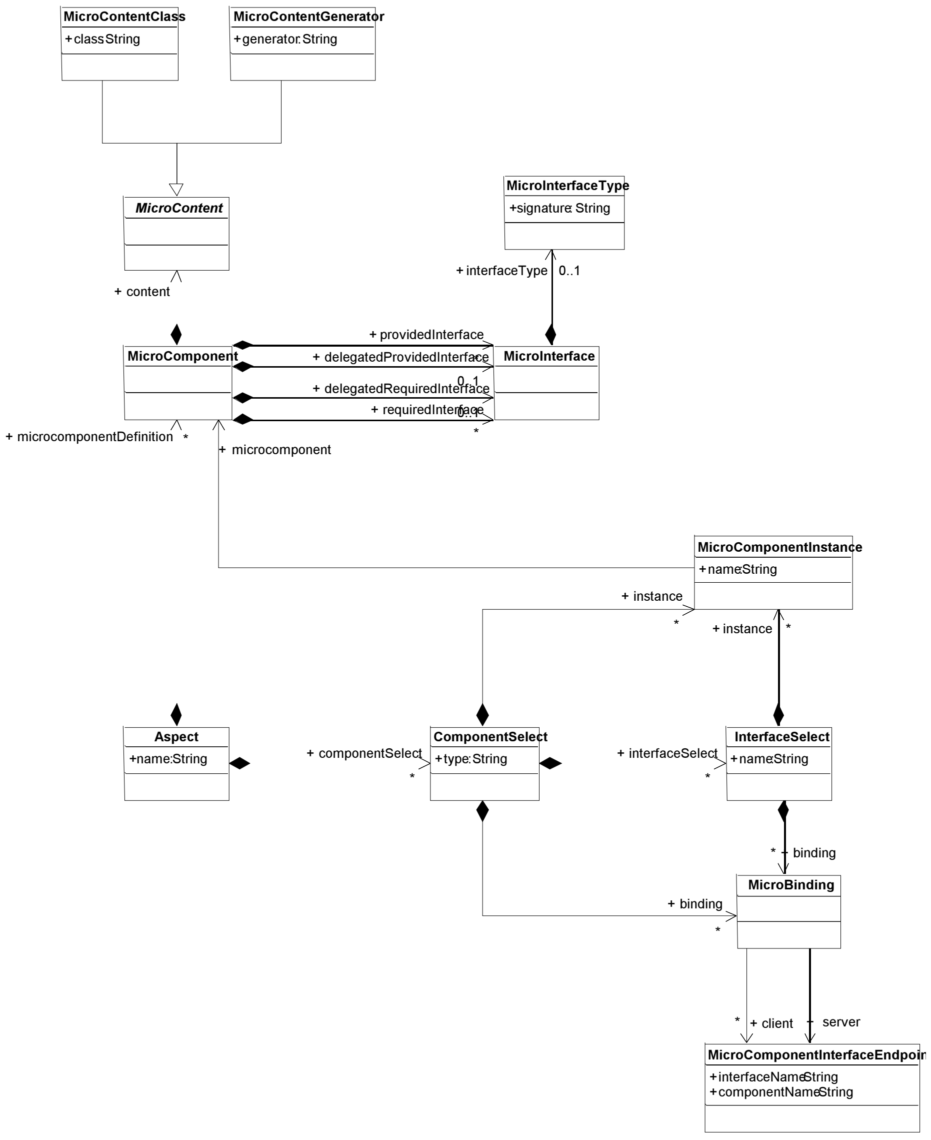
1. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation, Proceedings of ICSE 2002, Orlando, USA, May 2002
2. Allen, R.: A Formal Approach to Software Architecture, PhD thesis, School of Computer Science, Carnegie Mellon University, 1997
3. Bures, T., Plasil, F.: Communication Style Driven Connector Configurations, In Software Engineering Research and Applications, LNCS3026, 2004
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J. B.: An Open Component Model and Its Support in Java, Proceedings of CBSE 2004, Edinburgh, UK, May 2004
5. Chappell, D. A., Enterprise Service Bus, O'Reilly Media, Jun 2004
6. Dashofy, E. M., van der Hoek, A., Taylor, R. N.: A highly-extensible, XML-based architecture description language, Proceedings of WICSA'01, Amsterdam, Netherlands, August 2001
7. Enterprise Java Beans specification, version 2.1, Sun Microsystems, November 2003
8. Hnětynka, P., Píše, M.: Hand-written vs. MOF-based Metadata Repositories: The SOFA Experience, Proceedings of ECBS 2004, Brno, Czech Republic, IEEE CS, May 2004
9. Hnětynka, P., Plášil, F.: Distributed Versioning Model for MOF, Proceedings of WISICT'04, Cancun, Mexico, January 2004
10. Inverardi, I., Wolf, A.L., Yankelevich, D.: Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model, IEEE Transactions on Software Engineering, vol. 21, no. 4, April 1995
11. Iribarne, L.: Web Components: A Comparison between Web Services and Software Components, Colombian Journal of Computation, Vol. 5, No. 1, Jun 2004
12. Julia, http://forge.objectweb.org/projects/fractal/
13. Lau, K.-K., Wang, Z.: A Taxonomy of Software Component Models, Proceedings of EUROMICRO-SEAA'05, Porto, Portugal, Sep 2005
14. Magee, J., Kramer, J.: Dynamic Structure in Software Architectures, Proceedings of FSE'4, San Francisco, USA, Oct 1996
15. Medvidovic, N.: ADLs and dynamic architecture changes, Joint Proceedings SIGSOFT'1996 Workshops, ACM Press, New York, USA, Oct 1996
16. Mencl, V., Bures, T.: Microcomponent-Based Component Controllers: A Foundation for Component Aspects, Proceedings of APSEC 2005,Taipei, Taiwan, IEEE CS, December 2005
17. Object Management Group: CORBA Components, v 3.0, OMG document formal/02-06-65, June 2002
18. Object Management Group: Deployment and Configuration of Component-based Distributed Applications Specification, OMG document ptc/ 05-01-07, January 2005
19. Object Management Group: MOF 2.0 Core, OMG document ptc/03-10-04, October 2004
20. Plášil, F., Bálek, D., Janeček, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, USA, IEEE CS, May 1998
21. Plášil, F., Višňovský, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, November 2002

22. SOFA prototype, http://sofa.objectweb.org/
23. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, January 2002
24. Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead, E. J., Robbins, J. E., Nies, K. A., Oreizy, P., Dubrow, D. L.: A Component- and Message-Based Architectural Style for GUI Software, IEEE Transactions on Software Engineering, Vol. 22, No. 6, Jun 1996
25. WebServices, http://www.w3.org/2002/ws/
26. Wells, G.: Coordination Languages: Back to the Future with Linda, Proceedings of WCAT'05, Glasgow, UK, Jul 2005
27. Wermelingera, M., Fiadeiro, J. L.: A graph transformation approach to software architecture reconfiguration, Science of Computer Programming, Volume 44, Issue 2, August 2002

Appendix A


Property
+ type :String
+ property
+ property
NamedEntity
+ name :String
Feature
+value : String
+ communicationFeature
Version
+ version
VersionedEntity
<< enumeration >>
ConnectionType
+ normal :
+ utility :
InterfaceType
+signature :String
+ interfaceType
Interface
+communicationStyle :String
+connectionType :ConnectionType
+isCollection : boolean
+isOptional : boolean
+ providedInterface
+ returnInterface
+ requiredInterface
MappedProperty
+ mappedProperty
Frame
+protocol :String
+ instantiatesFrame
0..1
1..*
+ implements
xor
+ instance
+ anotation
Annotation
+ anotation
Architecture
+ subcomponent
SubcomponentInstance
0..1
+ instantiatesArchitecture
+
subcomponent
+ binding
Binding
<< Singleton >>
Factory
<< Singleton >>
TopLevel
Delegation
Subsumption
Connector
+ inEndpoint
+ endpoint
+ outEndpoint
+ outEndpoint
ComponentInterfacesEndpoint
+interfaceName :String
SubcomponentInterfaceEndpoint
+interaceName :String
+
inEndpoint

Appendix B


MicroContentClass
+class:String
MicroContentGenerator
+generator:String
MicroContent
MicroInterfaceType
+signature: String
+interfaceType
0..1
+ content
+ providedInterface
+ delegatedProvidedInterface
+ delegatedRequiredInterface
+ requiredInterface
0..1
*
MicroComponent
MicroInterface
+ microcomponentDefinition
+ microcomponent
MicroComponentInstance
+name:String
+ instance
Aspect
+name:String
+ componentSelect
ComponentSelect
+type:String
+ interfaceSelect
InterfaceSelect
+name:String
+ binding
MicroBinding
+ client
+ server
MicroComponentInterfaceEndpoint
+interfaceName:String
+componentName:String