# SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model

Tomáš Bureš[2,3], Petr Hnětynka[1,2], František Plášil[2,3]

[1]*Performance Engineering Lab School of Computer Science and Informatics University College Dublin Ireland*

[2]*Department of SW Engineering Faculty of Mathematics and Physics Charles University in Prague Czech Republic*

[3]*Academy of Sciences Institute of Computer Science Czech Republic*

*{bures, hnetynka, plasil}@nenya.ms.mff.cuni.cz*

## Abstract

*Component-based software engineering is a powerful paradigm for building large applications. However, our experience with building application of components is that the existing advanced component models (such as those offering component nesting, behavior specification and checking, dynamic reconfiguration to some extent, etc.) are subject to a lot of limitations and issues which prevent them from being accepted more widely (by industry in particular). We claim that these issues are specifically related to (a) the lack of support for dynamic reconfigurations of hierarchical architectures, (b) poor support for modeling and extendibility of the control part of a component, and (c) the lack of support for different communication styles applied in inter-component communication. In this paper, we show how these problems can be addressed and present an advanced component system SOFA 2.0 as a proof of the concept. This system is based on its predecessor SOFA, but it incorporates a number of enhancements and improvements.*

## 1. Introduction

Over the past few years, the component-based development (CBD) [25] has been recognized as a viable way of building software systems. Although there are many different views on what a component is and what its features are, common consensus regards a component as a black-box entity with well defined interfaces and behavior, and emphasizes, as one of the key features, its reusability in different contexts without any need of knowing or modifying component's internals. From a design view, components – especially hierarchical ones – can be viewed as gray-box/glass-box entities with the internal structure visible as a set of communicating subcomponents.

The set of rules defining components' creation, composition, life-cycle, and other features are usually referred to as a *component model*. The reification of a component model in a particular implementation (and runtime environment) is called component system/platform. As there is no general agreement on detailed features of components, we assume the concept of component is to be always interpreted only within a particular component model.

The idea of component-based development has already taken shape in a number of component systems – both industrial and academic. The industrial systems (represented mainly by EJB [9] and CCM [18]) are oriented on providing a stable and mature runtime, even at the cost of sacrificing the option of building component hierarchies and other advanced features (such as multiple communication styles, behavior description, etc.). On the other hand, the academic component models provide a rich set of features, especially allowing for powerful design with the help of hierarchical architectures, behavior specification, different communication styles, etc. Their main flaw typically lies in the insufficient runtime support – many of the academic systems are only design-oriented, without any sort of runtime environment [3,15].

In our view, the reason for such a big gap between the industrial models and academic component models lies in the fact that it is very difficult to properly balance the semantics of advanced features, so that they can be well grasped at design time, and correctly and flawlessly employed at runtime.

Newer than CBD, but quickly spreading are service-oriented architectures (SOA). SOA-based systems are already commonly used in industry (e.g., WebServices [26]). In our view, the reason for the success of SOA lies in the fact that SOA in a sense lies halfway between the simple flat industrial component models and the hierarchical and complex academic component models. A service in SOA has also well defined interfaces and a new service can be built by reusing some of the existing services. However, SOA lacks a first-class concept of nested service. New services are formed by specifying the interplay of services being reused, without any hierarchical nesting (as opposed to CBD). Since there is no strong concept of encapsulation (typically achieved by nesting), it is very difficult to deploy and reuse a service in different contexts.

### 1.1. Goals and structure of the paper

In this paper, we use our experience with designing and implementing the SOFA component system [21,24] and also our experience with building component applications for the SOFA and Fractal [4] component systems [8].

The goals of the paper are twofold: First, we analyze strong and weak points of component systems based on a hierarchical component model, and afterwards, we describe a new component system (SOFA 2.0) featuring a hierarchical component model and many other novel advanced features based on the lessons we learned with writing non-trivial applications for SOFA and Fractal. SOFA 2.0 aims at balancing these features and integrating them to form a coherent model, which can be also consistently reified at runtime.

The structure of the paper is as follows. On the basis of the SOFA component system, Section 2 analyzes the main limitations of the contemporary component-based systems with a hierarchical component model and Section 3 presents the SOFA 2.0 design and implementation. Section 4 contains evaluation and related work, while Section 5 concludes the paper.

## 2. Issues of hierarchical components

SOFA is a typical academic component based system. It uses a hierarchical component model with components being 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. Frame is a black-box view of a component. It defines the component's provided and required interfaces. Architecture is a gray-box view of a component; it implements the component's frame by specifying the subcomponents and their interconnections on the first level of nesting. Components are interconnected via bindings among interfaces. All bindings are furnished via connectors [6], which are first class entities (like components). Behavior of SOFA components can be captured formally via behavior protocols [22].

Runtime structure of a component is composed of a control part, which consists of the component manager, and a functional part, which in the case of a primitive component consists of code of the component and, in the case of a composite component, of other subcomponents.

Development lifecycle of a SOFA component is quite similar to other component systems. First, an ADL specification has to be written. The specification is then used to generate skeletons of the component implementation. The developer implements the primitive components and inserts them into a repository. In order to launch a component-based application, it is necessary to prepare a deployment plan, where components are assigned to particular host computers and resources are allocated. Finally, according to this plan, the application is deployed and launched.

Although SOFA has been a very innovative and promising platform for building large software systems, its usage revealed several limitations and obstacles. Interestingly, these issues are not SOFA-specific, since other component systems suffer of similar problems, namely (1) a limited support for dynamic reconfigurations, (2) no structure of the control part of a component and (3) unbalanced support for multiple communication styles. In the rest of this section, we describe each of them in more detail.

### 2.1. Dynamic reconfigurations

By *dynamic reconfiguration* we mean a runtime modification of the application architecture, i.e., adding and removing components at runtime, passing references to components, etc. The problem of dynamic reconfigurations lies in the fact that it is very difficult to describe the dynamicity of an application at design time. A naive solution to the problem by forbidding dynamic reconfiguration is not feasible, since dynamic changes of an architecture are inherent to many applications [16]. The other extreme – neglecting dynamic reconfigurations at design time and allowing for arbitrary ones at runtime – is not appropriate either. It leads to an uncontrolled modification of the architecture (evolution gap [11]), which is inherently error-prone. Thus, it is a necessity to reflect runtime reconfiguration at the design time. Also, based on our experience with non-trivial case studies, we regard this issue to be one of the primary hindrances of the wider usage of hierarchical component models.

### 2.2. Control part of components

In addition to *business* interfaces (i.e., the “classical” provided and required interfaces), components usually feature *control* interfaces. From architectural view, these are the provided interfaces, which correspond to non-functional features of components, i.e., life-cycle management, reconfiguration, introspection, etc – controllers in general. Even though the functionality implemented by controllers does not have to be explicitly accessible via public component interfaces, it is typically present in all component systems.

Since controllers provide access to non-functional aspects of components, they should not be freely accessible from the code of components (i.e., the application’s business logic) – controllers are intended to be used by runtime environment and administration and deployment tools. However, our experience with non-trivial component applications indicates that the code implementing component’s business logic should be aware of the fact that it is a part of the component and it should have means to access some limited and specifically tailored functionality of the component’s controllers (e.g., accessing component properties, signalizing a quiescent state during dynamic update).

The existing component models having support for controllers (such as Fractal) do not further elaborate on structuring and extending the controllers. However, we claim (and our experience confirms so) that explicit modeling of the control part and a simple extension mechanism may be of a great asset, especially when trying to extend the core functionality of the runtime environment [17].

### 2.3. Multiple communication styles

Communication style is a paradigm which components use for communication. The idea of different communication styles was coined in [23]. There, every architectural style gives a specific semantics to the concepts of component and connector (e.g., in pipe-and-filter architectural pattern, we interpret filter processes as components and Unix-like pipes as connectors). In order to support different architectural styles in component systems, we have to generalize the concept of component binding and explicitly capture it by a first class entity – connector. The communication style then defines the functionality of each connector. In SOFA, we have distinguished four communication styles – method invocation, message passing, streaming, and distributed shared memory [6].

The advantage of supporting different architectural styles lies not only in easier and more comprehensible design, but it even manifests itself at runtime: From the knowledge of the communication style, the inter-component communication can be optimized by choosing an appropriate middleware (e.g., CORBA for remote method invocation, TCP/IP for bi-directional streaming, etc.) for each binding.

Communication style strongly influences the way components can be bound together, allowing even for connecting a required interface to another required interface (e.g., in the case of TCP/IP streaming). Thus, without an explicit support for communication styles, it is very difficult to model different architectural styles and benefit at runtime from their knowledge at design time. When such support is not present, components with middleware functionality have to be typically employed (e.g., a component implementing a TCP/IP socket), which however spoils the design by mixing business and communication logic and makes other features (e.g., behavior checking or dynamic update) very complicated.

Although these results of the software architecture research have been around for a number of years, they have not been sufficiently adopted by the major component based systems (e.g., EJB, CCM, Fractal), which still rely only on remote method invocation and optionally, also on message passing (e.g., CCM). From this point of view, SOFA has been an innovative component system. However, still the support for multiple communication styles is unbalanced in SOFA and not well integrated with other SOFA abstractions.

## 3. SOFA 2.0

In this section, we provide a detailed description of the SOFA 2.0 component system and its component model. This new component system is based on our experience with SOFA. In SOFA 2.0, we introduce the new concepts which we felt were missing in SOFA (e.g., the dynamic reconfiguration, explicit controller) and we have also improved several concepts already existing in the original SOFA (e.g., multiple communication styles, deployment).

On the other hand, the main concepts and general design of the SOFA system remained the same. The first-class entities are still components and connectors; and SOFA 2.0 still employs a hierarchical component model.

In the original SOFA system, the semantics of key abstractions was defined together with the ADL language specification. In the SOFA 2.0 system, however, we use a meta-model based definition. More specifically, we use the MOF technology [20] for designing the component model (a meta-model in the terms of MOF). Such approach has many advantages – automated generation of a repository with standardized interface, standardized XML-based interchange format, support for automated generation of models' editors, etc. (for details please refer to [12]). This meta-model directly serves for component's specification (i.e., it is on the same level as ADL). The specification is then used at development time for generating code skeletons for primitive components, at the deployment time to prepare a deployment plan, and at execution time to actually set up an application.

As an implementation language for generated code fragments and implementation of primitive components, we use Java since it provides a rich set of advanced features (e.g., easy dynamic class-loading, introspection, etc.). However, the meta-model and SOFA 2.0 abstractions are programming language independent.

## 3.1. SOFA 2.0 component model

In this section, we describe the meta-model of SOFA 2.0 components. Throughout the text, the terms in italic are elements of the meta-model (its complete diagram is in Appendix A).

**3.1.1. Common elements.** There are several common elements (*NamedEntity*, *VersionedEntity*, and *Version*) used thorough the whole meta-model. We start the overview of the meta-model by a short description of these elements.

The *NamedEntity* class[1] is used as an ancestor of all elements having a name (the *name* attribute). *VersionedEntity* serves as a base-class for all entities, which may exist in several versions distinguished by a version number (via the class *Version*). Due to space constraints, we do not elaborate in this paper on the actual versioning scheme used in SOFA 2.0 and leave the *Version* class unspecified (for more details on versioning in SOFA please refer to [13]).

**3.1.2. Component frame.** The core element of the component frame abstraction is the *Frame* class, which defines the black-box view of a component. The provided and required interfaces of the frame are defined via *Frame's* associations with the *Interface* class. This class defines the name of an interface (by inheriting from *NamedEntity*), the type of the interface (by referring to *InterfaceType*). The *InterfaceType* class is a separately defined element (i.e., it exists in a model by its own) defining the type of the interface by the means of *signature*, which is a reference to an interface definition in an underlying language (Java in our case). Moreover, it inherits from *VersionedEntity* to allow versioning of interfaces. The class *Interface* contains binding-oriented attributes *connectionType*, and *isCollection*. The *connectionType* attribute can be either *normal* or *utility* – it determines whether an interface can be used in the utility reconfiguration pattern (Sect. 3.2.1). The *isCollection* attribute captures the cardinality of an interface – either the interface can participate in just a *single* binding or in a number of bindings (cardinality *multiple*). Interface further defines *communicationStyle* and *communicationFeatures* (represented by the *Feature* class). A communication style (see Sect. 2.3) denotes the communication paradigm (e.g., method invocation, streaming, etc.) that is expected by the associated component. Communication features then allow further refining of the communication style by specifying non- and extra-functional properties (e.g., that sensitive data are transmitted, etc.). The communication style and communication features are used at deployment time as a source of information for the connector generator (Sect. 3.2.3). Finally, an interface can be annotated by sub-classes of *Annotation* – particularly by the *Factory* annotation, meaning that the interface is a factory dynamically creating new components (Sect. 3.2.1).

The remaining elements associated with the *Frame* class are *Annotation* and *Property*. Through *Annotation* – particularly the *TopLevel* annotation, a frame can be marked as the top-level frame in the application (representing the whole application). *Property* allows to define the configuration properties of a component, which can be set up at deployment time.

**3.1.3. Component architecture.** The frame of a component is implemented by an A*rchitecture*, which represents a gray-box view of the component. A single frame can be implemented by several architectures, and also a single architecture can implement several frames. It is similar to object-oriented programming, where a single class can implement several interfaces.

Thus the Architecture class features the *subcomponent* association with *SubcomponentInstance*. If the set of subcomponents is empty, then the architecture refers to a primitive component (directly implemented in a programming language). In the opposite case, each *SubcomponentInstance* refers either to the *Frame* or to another *Architecture* (in the

[1] *NamedEntity* (like all other element of the meta-model) is a meta-class and as such it should be marked with meta- prefix. However, for the sake of better readability we omit this prefix in the text.

meta-model, this fact is emphasized by the comment with the *xor* label). By referring to another architecture, complex architectures can be built, specifying multiple levels of architecture nesting (as opposed to the original SOFA).

Connections among subcomponents are represented by the *Binding* class, or, more specifically, by one of its subclass – *Delegation*, *Subsumption*, and *Connector*.

The first two classes allow "forwarding" of component interfaces to subcomponents. *Delegation* connects a provided interface of the component to one of its subcomponent's provided interface and *Subsumption* connects a subcomponent's required interface to a required interface of the component. The last class – *Connector* – represents a connection between two or more subcomponents. In all three cases, a particular connection is described via the appropriate combination of endpoints – the *ComponentInterfaceEndpoint*, which is a plain connection end pointing to the component interface and *SubcomponentInterfaceEndpoint*, which is a connection end pointing to the subcomponent interface. The important aspect of this meta-model is that it allows plain connections not only between the provided and required interfaces but also provided-to-provided and required-to-required connections. This feature helps smoothly integrate of multiple communication styles.

The remaining characteristics of *Architecture* are the association with the *Properties* class, introduced to describe component properties at the architecture level, and the association with the *MappedProperty* class. This association makes subcomponents' properties visible also as the properties of the parent component.

## 3.2. Runtime structure

**3.2.1. Dynamic reconfiguration of components.** By *dynamic reconfiguration* we mean a run time modification of an application's architecture. A special case of dynamic reconfiguration is *dynamic update* of a component, i.e., replacing a particular component by another one having compatible interfaces. Dynamic update is easy to handle, because all the changes are local to the updated component, being thus transparent to the rest of the application. A dynamic update is a "real" dynamic reconfiguration because the new component can have a completely different internal structure. A general dynamic reconfiguration is an arbitrary modification of an application architecture though.

To prevent an uncontrolled modification of an architecture (the evolution gap problem – Sect. 2.1), in SOFA 2.0, we only permit those dynamic reconfigurations done in accordance to a predefined *reconfiguration pattern*. Currently, we allow the following three reconfiguration patterns: (i) nested factory, (ii) component removal, and (iii) utility interface. All these patterns and reasons for choosing them are in detail described in [11].

The most important consequence of introducing the utility interface concept is that it has brought some features of SOA systems into component-based models. Such feature integration allows taking advantages of both these methodologies (e.g., encapsulation and hierarchical components of component models and simple dynamic reconfiguration inherent to service-oriented architectures).

**3.2.2. Controllers.** The control part of a component in SOFA 2.0 is modular and extensible. The general idea of this approach and its application to Fractal component model is described in [17]. The control part of a component is in our approach modeled as composed of *microcomponents*. The microcomponent model is a very minimalist one – it is flat (i.e., no nested microcomponents) featuring no connectors and no distribution. Additionally, to avoid recursion, a microcomponent does not have any extensible or structured control part. In principle, a microcomponent is just a class implementing a specified interface.

On the top of the microcomponent model we define *aspects* as consistent extensions of the control part. An aspect comprises a definition of microcomponents and of microcomponent instantiation patterns. By applying a number of aspects, a control part with the desired functionality is obtained. The aspects to be applied are specified at deployment time.

There is a core aspect in SOFA 2.0, which is present in all controllers. This core aspect introduces the control interfaces of a *lifecycle* controller (starting/stopping/ updating a component) and a *binding* controller (adding/removing connections among components), and provides basic functionality of these controllers.

**3.2.3. Connectors.** In SOFA 2.0 we distinguish two types of connectors – design and runtime. Design connectors were used in the SOFA component model, where they take the form of hyperedges connecting several component interfaces. The specification of a design connector consists of the communication styles and communication-related features associated with each component interface involved in the communication.

Runtime connectors are the code artifacts used at runtime to implement the design connectors. In our approach, we use a connector generator [5] to automatically (i.e., without human assistance) create runtime connectors from their design counterparts. The generation is performed at deployment time, just before the application is prepared to be launched. Since the generation is postponed to such a late stage, it benefits from the complete knowledge of the target deployment environment (i.e., capabilities of the host computers, capabilities of the network, etc.) so that an accordingly optimized connector code can be generated.

**3.2.4. Architecture of the runtime environment.** A SOFA 2.0 application is executed in the distributed runtime environment called SOFAnode, which consists of a number of deployment docks. A deployment dock is a component container (i.e., Java virtual machine plus SOFA 2.0 runtime) hosted on a particular computer and providing runtime functionality for executing components. An application can span several deployment docks within one SOFAnode. The assignment of components to particular deployment docks is done during deployment. A binding heading to an frame outside of a SOFAnode is possible via a utility interface, which can be bound to an already running external service or be exposed as a service (both via the SOA paradigm). A reference to the required service is supplied during deployment.

The implementation of a component is formed by a number of Java classes realizing its functional part and the microcomponents of its control part (see Sect. 3.2.2). If the component is composite, its implementation is composed just of classes realizing microcomponents.

Apart from the deployment docks, a SOFAnode contains also a repository, which holds components' description (i.e., information about component frames, architectures, and interface types) and their implementations (i.e., the code of primitive and composite components and the generated code of connectors). Thus, the repository is used throughout the whole application lifecycle as the central source of components' descriptions as well as a code base.

## 4. Evaluation and related work

*Evaluation:* The paper presents the new version of the SOFA component system, called SOFA 2.0. This new version aims at removing several limitations of the original version of SOFA – mainly the lack of support of dynamic reconfigurations of an architecture, well-structured and extensible control part of a component, and multiple communication styles among components.

The newly designed dynamic reconfiguration strategy is based on asking compliance with well defined reconfiguration patterns. These patterns also include the utility interface pattern, which brings into a component-based system several features of SOA. In an extreme case, provided most of the "components" are external services, it allows almost general dynamic reconfiguration. The control part of a component is composed of microcomponents, which allows smooth and simple extensibility of the control functionality of the component. The support for multiple communication styles is available not only at design time but also at runtime. All the proposed enhancements are based on our experience with non-trivial component applications [8] developed for the SOFA and Fractal component systems.

In our view, an important step towards evaluating component systems has been taken by Lau et al. [14]. They define a taxonomy of component models using the criteria of component composition at different stages of component lifecycle (design and deployment in particular). This taxonomy is used to classify a number of existing component systems. The original version of SOFA has been classified as belonging to the most advanced category (together with Koala and KobrA) mostly due its ability to (i) compose components at design time, (ii) store composed components in a repository, and (iii) reuse the stored components (including composite ones) in further compositions. The only flaw ascribed to the original SOFA is its inability of composition at deployment time. But in our view and also according to [19], no composition should happen at deployment time (only assigning components to concrete nodes, allocating resources, etc.). The "deployment time" phase described in [14] in fact corresponds to runtime phase, during which we address the composition by using the patterns for dynamic reconfiguration. Thus, in this view, SOFA 2.0 meets all the criteria imposed by [14].

*Related work:* Below is an overview of component systems and models, which are contemporarily used and/or developed. We describe each of them together with its advantages and drawbacks. At the end of this survey, we also briefly focus on the area of SOA systems and their relation to component models.

The CORBA Component Model (CCM) [18] and Enterprise Java Beans (EJB) [9] are representatives of the industrial component systems employing flat component model. CCM components' interfaces are divided into provided and required ones; in addition, interfaces for synchronous and asynchronous invocations are distinguished. Components are defined in IDL (Interface Description Language), but IDL does not provide any support for describing component composition/architecture. EJB components are defined directly in Java. They do not have explicitly specified required interfaces. Every EJB component type has a *home* interface, through which new components can be created. A similar concept in CCM is called *factory* interface. Both these interfaces can be seen as controller interfaces. In addition, CCM components can have *attributes*, which are named values exposed via getter and setter methods and primarily intended for component parameterization. These attributes can be seen as determining alternatives of the control functionality of the component. In both systems, these control interfaces are specified firmly, not being extensible in any way.

Fractal [4] uses also a component model with hierarchically nested components. In Fractal, each component can have multiple control interfaces. The number and types of the control interfaces depend on the configuration of a run-time environment. Each component is divided into 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.

A rather different approach to component systems is taken by ArchJava [1] and Java/A [2]. They are both Java extensions introducing concepts of components, bindings, and architectures at the language level. Both of them restrict the dynamic reconfigurations to an extent – ArchJava by forcing the developer to enumerate the "allowed" connections, and Java/A by the means of "architecture templates". The common motivation of ArchJava and Java/A is to address to the problem of *architecture erosion* (i.e., the decisions taken during development are reflected only in the code and not propagated back to the architecture). However, this problem cannot occur in SOFA 2.0 as a component bears its identity by referring to its architecture through the whole development lifecycle (from design to runtime). Thus, an attempt to instantiate a component, which code does not conform to its architecture, results in a failure.

The key representatives of design-oriented component models are Acme [10], Darwin [15] and Wright [3]. All of them provide an ADL language for modeling and analyzing component architectures.

Acme is a generic architecture description language developed at Carnegie Mellon University. It aims at serving as a common representation of software architectures. It permits integration of diverse collections of independently developed architectural analysis tools. Acme is supports four distinct aspects of architecture - structure, properties, design constraints, and types and styles. From the structural point of view, architectures in Acme can be nested and connectors are regarded as first-class entities. The properties provide a way of associating auxiliary information defining the run-time semantics of a system, interaction protocols, scheduling constraints, resource consumption, etc. The design constraints determine how an architectural design is permitted to evolve at runtime. To specify the constraints, a language based on first order predicate logic is used. Eventually, the types and styles aspect provide means to package the above mentioned specifications into families of systems (e.g., pipe and filter family).

Darwin uses a component model with hierarchically nested components, with provided and required interfaces. Connections among components are plain bindings; no connectors are considered. Darwin allows dynamic reconfiguration via lazy and direct dynamic instantiation of components. As to the lazy dynamic instantiation, a component with a provided interface is not instantiated until the first usage of such an interface. Direct dynamic instantiation allows defining architectures that dynamically evolve in an arbitrary way but new connections are not explicitly captured on the architecture level.

Wright also uses a component model with nested components. A component specification is composed of two parts – interface and computation; interface further consists of ports (a port is an interface as understood in other component models). Ports define separate interactions in which the component will participate and computation, defined in a CSP-like notation, defines the behavior of the component. Interconnections among component interfaces are made through connectors. A connector is formed of a glue and roles. The roles are the endings of the connector that are directly tied to component interfaces; the glue describes the behavior of the connector as sequences of events and its roles together. Like computation, glue is also described in a CSP-like notation.

Though powerful in design, none of Acme, Darwin, or Wright specifies any runtime environment for component execution.

Software services [26] and SOA are concepts very similar to components and CBD. Software services are specified as business interface descriptions (there are no control interfaces). Moreover, the distinguishing of multiple communication styles is not an issue, since the services are supposed to be accessed via messaging. Also, for SOA systems, dynamic reconfiguration is not an issue either because an application composed of services does not employ any rigid architecture and the composition of services (the service interplay) is rather specified per request; the specification is typically based on approaches like coordination languages [27] and routing of messages [7].

## 5. Conclusion

In this paper, we have shown the limitations of contemporary component-based systems and have presented a new version of the SOFA component system – SOFA 2.0 – that aims at targeting all these limitations. The artculation of the limitations is based on our experience with building large components applications. Currently, we are in the process of implementing SOFA 2.0; we have already specified all the necessary meta-models and are implementing the SOFA 2.0 runtime. A working implementation is expected within several months.

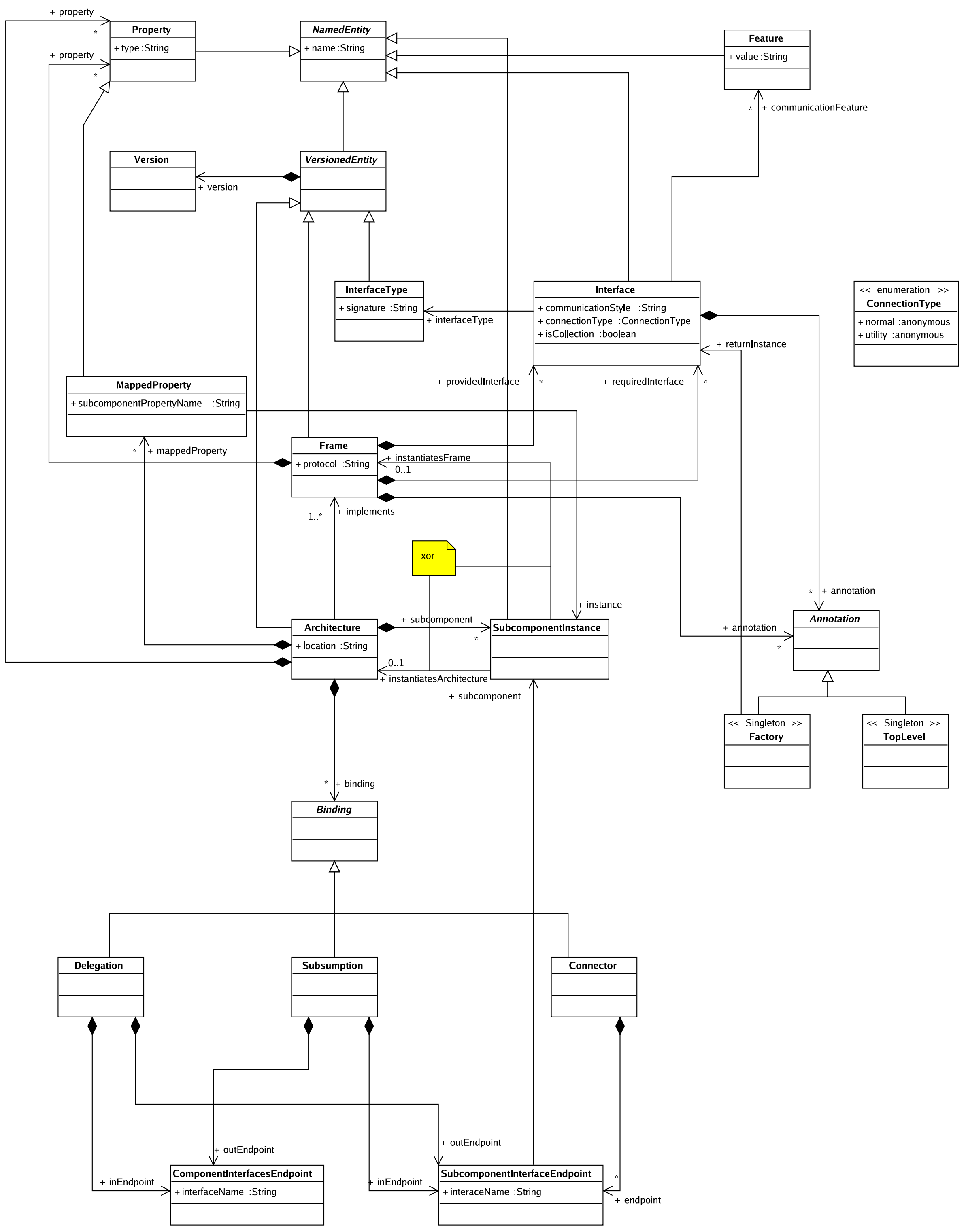
## Acknowledgements

The authors would like to thank Vladimír Mencl and Lucia Kapová for valuable comments and Jan Klesnil, Ondřej Kmoch, Tomáš Kohan, and Pavel Kotrč for contributing to meta-model design. This work was partially supported by the Grant Agency of the Czech Republic project 201/06/0770. The support of the Informatics Commercialisation initiative of Enterprise Ireland is gratefully acknowledged.

## References

1. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation, Proc. of ICSE 2002, Orlando, USA, May 2002
2. Baumeister, H., Hacklinger, F., Hennicker, R., Knapp, A., Wirsing, M.: A Component Model for Architectural Programming, Proc. of FACS'05, Macao, Oct 2005
3. Allen, R.: A Formal Approach to Software Architecture, PhD thesis, School of Computer Science, Carnegie Mellon University, 1997
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J. B.: An Open Component Model and Its Support in Java, Proc. of CBSE'04, Edinburgh, UK, May 2004
5. Bureš, T.: Automated Synthesis of Connectors for Heterogeneous Deployment, Tech. Rep. No. 2005/4, Dep. of SW Engineering, Charles University, Prague, Aug 2005
6. Bureš, T., Plášil, F.: Communication Style Driven Connector Configurations, In Software Engineering Research and Applications, LNCS3026, 2004
7. Chappell, D. A., Enterprise Service Bus, O'Reilly Media, Jun 2004
8. Component Reliability Extensions for Fractal Component Model, http://kraken.cs.cas.cz/ft/public/public_index.phtml
9. Enterprise Java Beans specification, version 2.1, Sun Microsystems, Nov 2003
10. Garlan, D., Monroe, R. T., Wile, D.: Acme: Architectural Description of Component-Based Systems, Foundations of Component-Based Systems, Cambridge University Press, 2000
11. Hnětynka, P., Plášil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models, Accepted for publication in Proc. of CBSE 2006, Vasteras near Stockholm, Sweden, Jun 2006
12. Hnětynka, P., Píše, M.: Hand-written vs. MOF-based Metadata Repositories: The SOFA Experience, Proc. of ECBS 2004, Brno, Czech Republic, May 2004
13. Hnětynka, P., Plášil, F.: Distributed Versioning Model for MOF, Proc. of WISICT'04, Cancun, Mexico, Jan 2004
14. Lau, K.-K., Wang, Z.: A Taxonomy of Software Component Models, Proc. of EUROMICRO-SEAA'05, Porto, Portugal, Sep 2005
15. Magee, J., Kramer, J.: Dynamic Structure in Software Architectures, Proc. of FSE'4, San Francisco, USA, Oct 1996
16. Medvidovic, N.: ADLs and dynamic architecture changes, Joint Proc. SIGSOFT'1996 Workshops, ACM Press, New York, USA, Oct 1996
17. Mencl, V., Bureš, T.: Microcomponent-Based Component Controllers: A Foundation for Component Aspects, Proc. of APSEC 2005, Dec 2005
18. OMG: CORBA Components, v 3.0, OMG document formal/02-06-65, Jun 2002
19. OMG: Deployment and Configuration of Component-based Distributed Applications Specification, OMG document ptc/ 05-01-07, Jan 2005
20. OMG: MOF 2.0 Core, OMG document ptc/06-01-01, Jan 2006
21. Plášil, F., Bálek, D., Janeček, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proc. of ICCDS'98, Annapolis, USA, May 1998
22. Plášil, F., Višňovský, S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering, vol. 28, no. 11, Nov 2002
23. Shaw, M., Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, 1996
24. SOFA prototype, http://sofa.objectweb.org/
25. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, Jan 2002
26. WebServices, http://www.w3.org/2002/ws/
27. Wells, G.: Coordination Languages: Back to the Future with Linda, Proc. of WCAT'05, Glasgow, UK, Jul 2005

# Appendix A


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