Runtime Support for Advanced Component Concepts

Tomas Bures[1,2], Petr Hnetynka[1,3], Frantisek Plasil[1,2],
Jan Klesnil[1], Ondrej Kmoch[1], Tomas Kohan[1], Pavel Kotrc[1]

[1]Department of Software Engineering
Faculty of Mathematics and Physics,
Charles University, Malostranske namesti 25,
Prague 1, 118 00, Czech Republic

[2]Institute of Computer Science,
Academy of Sciences of the Czech Republic
Pod Vodarenskou vezi 2, Prague 8,
182 07, Czech Republic

[3]School of Computer Science and Informatics,
University College Dublin
Belfield, Dublin 4, Ireland

{*bures, hnetynka, plasil, klesnil, kmoch, kohan, kotrc*}*@dsrg.mff.cuni.cz*

Abstract

Component-based development has become a recognized technique for building large scale distributed applications. Although the maturity of this technique, there appears to be quite a significant gap between (a) component systems that are rich in advanced features (e.g., component nesting, software connectors, versioning, dynamic architectures), but which have typically only poor or even no runtime support, and (b) component systems with a solid runtime support, but which typically possess only a limited set of the advanced features. In our opinion, this is mainly due to the difficulties that arise when trying to give proper semantics to the features and reify them in development tools and an runtime platform. In this paper, we describe the implementation of the runtime environment for the SOFA 2.0 component model. In particular, we focus on the runtime support of the advanced features mentioned above. The described issues and the solution are not specific only to SOFA 2.0, but they are general and applicable to any other component system aiming at addressing such features.

1 Introduction

Component-based development (CBD) [27] has become a commonly used technique for building large scale enterprise systems. It has also found its way to other areas of software systems (e.g. GUI, embedded systems, product lines). In contrast to former development techniques, components allow for specifying not only the services provided by them but also services required from other components and/or environment. Thanks to this feature, components have brought easier reuse, integration, and rapid development of applications.

Even though, there are many views and definitions of what a component is, a general consensus is that a component is a black-box software entity with well defined interfaces and behavior. The set of all component features and rules for component lifecycle, composition etc. is usually called *component model*. From the composition point of view, component models can be divided into two categories — flat component models and hierarchical component models. In contrary to flat ones, the hierarchical component models allow forming composite components, i.e. the components hierarchically composed of other components. Thus, an application can be seen as tree of nested components.

The flat component models (e.g. EJB [14], CCM [19]) are typically relatively mature and quite heavily used in production of specific software applications, although they do not provide advanced features like multiple communication styles, composition verification, seamless distribution, etc. On the other hand, the hierarchical component models typically support such advanced features and concepts. However, the hierarchical models are mostly academic systems oriented only on design — i.e. they provide no or very limited runtime environment/platform (for instance they provide just an ADL compiler generating code fragments of components). In particular a runtime platform should provide a component repository, component container, basic services for components (typically in a form of libraries), and a user interface to control the runtime lifecycle of a component-based application. In our view, the main reason for such an imbalance is simply that it has not been yet found a way to properly make all such advanced features

coexist in a component model (and in its run time platform in particular).

As an attempt to address the imbalance between the rich set of features and the runtime and execution support, we have developed SOFA 2.0 component system in our group. SOFA 2.0 [5] is a component systems in which development we have drawn on our eight-year experience with designing, building and working with component systems. SOFA 2.0 offers a hierarchical component model together with advanced features like support for multiple communication styles, composition and behavior verification, clearly separated functional and control parts of components, seamless support for versioning and dynamic component updating, seamless distribution, runtime modification of architectures, support for SOA concepts, and extensible functionality of component containers.

The goal of this paper is to provide an overview of the issues we faced in the design and implementation of the SOFA runtime platform with respect to the aim to support all the advanced features mentioned above. We believe that these issues and their solutions are not specific only to SOFA 2.0, but that they are inherent to any component system aiming at addressing these advanced features.

This paper is a continuation of our work described in [5]. While the paper [5] has shown all features of the SOFA 2.0 component model and its global design, this paper is about the implementation and related issues.

The structure of the paper is as follows. Section 2 describes SOFA 2.0 in more depth. Section 3 details the implementation of all parts of SOFA 2.0. In Sect. 4, we present related work while Sect. 5 concludes the paper.

2 Overview of the SOFA 2.0 component model

SOFA 2.0 is a component model with a number of advanced features. The component model is formally specified using a meta-model which captures the concepts used in SOFA 2.0 and states relations among them (for the SOFA 2.0 meta-model, please refer to [5]).

In SOFA 2.0, a component is an encapsulated entity interacting with other components only via designated *provided* and *required* interfaces. A component can play the role of both a black-box and gray-box entity. The black-box role is reified in SOFA 2.0 as a *component frame*, which in fact represents a set of component interfaces, both provided and required, and determines the component's type. At the same time, it includes a specification of component behavior in terms of the event traces determined by the desired sequencing/parallel method call acceptance on the provided interfaces and their reactions on the required interfaces.

As a gray-box, a component is specified as an architecture that implements a particular component frame (or a number of frames). Being a hierarchical component model, SOFA 2.0 distinguishes two kinds of architectures (i.e. components) — *primitive* and *composite*. While a primitive architecture is in fact a direct implementation of the component in a particular programming language, a composite architecture is modeled as a composition of sub-components (i.e. other components). A composite architecture does not introduce any functional ("business") code; its functionality is determined by its sub-components and their composition (interface bindings). A composite architecture thus delegates calls on its own interfaces to some interfaces of its sub-components. A sub-component can be specified either by referencing its frame or by referencing another architecture (as an aside, it also references a frame). In principle, this way of frame-based sub-component definition introduces a variation point at which different implementations of component internals can be chosen at assembly time (Sect. 2.1).

As first-class entities, apart from components, SOFA 2.0 introduces also software connectors. This allows modeling explicitly component distribution and also employing different architectural and communication styles (e.g. pipe-and-filter, communication via a bus, sharing a memory) without the necessity of blurring an architecture by auxiliary components that actually realize a communication pattern. A connector specification determines a communication style and set of properties. The communication style reflects basic communication paradigm (e.g. RPC, asynchronous message delivery, streaming, shared memory) while the properties capture detailed requirements on the communication (e.g. that a particular security level is required).

Connectors are used to realize all tights (links) among component interfaces (i.e. delegations between a parent component and a sub-component and bindings among sub-components). A connector is modeled as a hyper-edge, which allows incorporating several component interfaces in one communication link (e.g, one server, several clients). It is possible to connect arbitrary interfaces together (e.g. required-to-required, provided-to-provided, etc.), which is especially useful when modeling for example communication via a bus (Fig. 1).

SOFA 2.0 allows for dynamic evolution of an architecture at runtime. Unlike other component systems, it pursues the idea of controlled evolution, which means that evolution has to conform to well-defined evolution patterns. This increases the manageability and predictability of an application's architectural evolution. In current SOFA 2.0, three evolution patterns are predefined: factory pattern, removal pattern, and service access pattern. As its name suggests, in factory pattern a designated component serves as a component factory [10]. The removal pattern serves for destroying of a component previously dynamically created. The

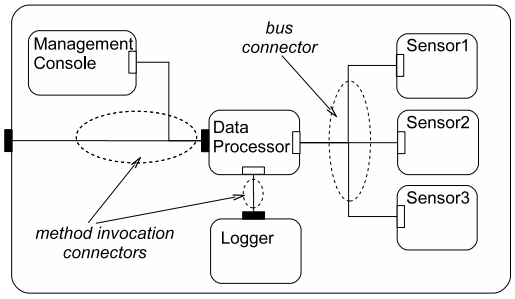

Figure 1. Different communication styles

employment of *utility interfaces* described below (allowing access to external services), comprises the service access pattern. Apart from design, SOFA 2.0 aims also at supporting components at runtime. This goal means that not only business functionality of components but also control functionality (e.g. managing the life-cycle, bindings, etc.) has to be addressed. SOFA 2.0 pursues clear separation of the control logic from the business logic, additionally it addresses the problem of the inextensibility of the control functionality, which is typically closely tied to a component system and cannot be easily changed or extended.

The solution addressing the control functionality is based on a dedicated micro-component model [18, 5], which is a very simple flat component model without any advanced features (distribution, separate control logic, connectors, etc.). It allows building a component's control part in a functionally modular manner via a composition of micro-components. Micro-components are organized to *component aspects*, each of which defines what micro-components to instantiate and how to incorporate them in the existing control part. A component aspect may also introduce a new *control interface* to provide another entry point to the control part of the component.

Although SOFA 2.0 is a component based system, it incorporates also a basic support for services and eventually allows for service-oriented architectures (SOA). SOFA 2.0 addresses two main tasks connected with services, namely (1) binding to and using a local or external service at runtime and (2) exposing a component interface as a service[1]. Basically, it allows a required interface to be bound to a service and a provided interface to be exposed as a service (e.g. WebService). Since there are inherently different rules for handling such interfaces, SOFA 2.0 allows for marking them as *utility interfaces* and it relaxes on some rules that hold for handling these interfaces — a required utility interface may be freely bound and unbound at runtime (this is typically based on a request to a service registry without the necessity to formally capture such dynamic behavior at design time), while a provided utility interface may exposed as a service and registered in a service registry, which makes it accessible to other component applications and even non-component-based clients.

[1] Please note that the concept of a local service is already heavily used in enterprise systems (e.g. obtaining a connection to a database, using JNDI registry, etc.).

2.1 Component lifecycle

Lifecycle of a SOFA 2.0 application involves several stages, namely (a) component development, (b) application assembly, and (c) application deployment and execution.

Development of a SOFA 2.0 application is quite straightforward, chiefly by composing already developed components available in the SOFA repository (Sect. 3.2.1). The development process starts with defining the architecture of an application. Using development tools, a developer can browse the repository content and compose existing components or build new ones. Then, the developer has to provide code of newly created primitive components. Finally, all new components are committed into the repository.

As an architecture is described mainly by frames, the next stage is an application assembly, when the frames (i.e. component types) are "refined" by particular architectures (i.e. component implementations). The process starts with the top-level component and recursively continues till primitive architectures. The result is specified by an assembly descriptor.

Finally, an assembled application is deployed, i.e. it is specified, where particular component of the application have to be executed and connectors are generated. The resulting information is captured in a deployment plan, which instructs the SOFA runtime how to execute the application.

3 Runtime environment

SOFA 2.0 is implemented in Java [9]. We have chosen Java because of its features such as platform independence, dynamic loading, type safety, and others. Another reason is that Java becomes more and more ubiquitous as devices like mobile phones, set-top-boxes embed the Java environment and applications for these devices are commonly written in Java. On the other hand, SOFA concepts are completely independent on used language and SOFA can be implemented in any procedural language.

A SOFA 2.0 runtime environment consists of several entities, which allow execution of SOFA applications. Moreover, a few of them also take part in development of components. The whole SOFA 2.0 environment is called *SOFAnode* and it consists of a number of *deployment docks* and a single *repository*. All these entities are described in detail in the following sections.

3.1 Components

From the implementation view, a SOFA 2.0 component is a set of Java classes and interfaces. The classes implement both the functional and control part of a component. Control part of it is formed of micro-components and in detail described in Sect. 3.3. As composite components are composed of other components, they have no direct functional part and therefore developers do not create any code for them. For a primitive component, there is no limitation about the number of classes, which implement it (obviously there has to be at least one implementing class).

SOFA 2.0 does not impose any particular requirement on classes implementing primitive components, i.e. there are no SOFA-specific interfaces/classes, which have to be implemented/extended. Instead, SOFA 2.0 uses an annotation-based approach (similar to the one described in [24]) where components' provisions, requirements, initializing methods, etc. are marked using annotations. An advantage of such an approach is that in code of components, there are no dependencies on the underlying platform and therefore, once developed, a component can be easily used in different component platforms. Another advantage of no extra dependencies is that implementation classes can be easily tested by tools like JUnit without starting the whole SOFA 2.0 environment.

Annotations are used directly by the SOFA 2.0 runtime environment and respective actions like setting requirements, obtaining provisions, and initializing components are performed using introspection. In a platform, where introspection cannot be used (e.g. SOFA 2.0 implementation for Java Micro Edition), we envision a tool which processes the annotations and prepares SOFA-specific code for the actions.

There are no limitation or constraints about components' own threads. For creating these threads, the component uses the regular Java API, i.e. the `Runnable` interface and `Thread` class. Internally (and completely transparent to component developers), each thread has assigned so called thread call context, which serves mainly for globally unique identification (across distributed deployment docks) of the thread and for holding call-related information like sessions, transaction IDs, etc.

3.2 SOFAnode

The *SOFAnode* is distributed SOFA 2.0 runtime environment, which consits mainly of the *repository* and set of *deployment docks*. These reside on (physical) deployment nodes (Fig. 2).

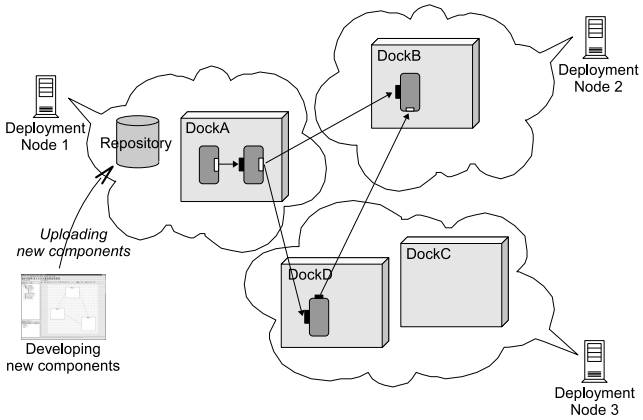

Figure 2. SOFAnode example

3.2.1 Repository

The *repository* is the heart of a SOFAnode. It stores both the meta-data about components as well as component implementations. The core of the repository is defined and generated using the EMF framework [13]. Over the core, there is developed an accessing layer, which allows remote access to the repository and provides helper methods simplifying its usage.

The repository is used as a storage of meta-data and code of components not only at run time but also at development time. All entities stored in the repository are versioned. In order to allow convenient development and usage of different versions of components, the repository supports *cloning*. When a developer starts a work on new components then he or she creates a new clone of the repository, which mirrors the whole content of the repository. Then, in the clone, the developer works on the new component (either creating completely new components or new versions of the existing ones) and finally, he or she merges the clone with the original repository. Using the cloning technique, the repository is kept in a consistent state; in the development clone, there can occur temporary inconsistencies (e.g. references to not yet existing components) but the original repository is still consistent and also the merging process ensures that only consistent clone can be committed. The repository cloning approach is inspired by the source configuration systems like GNU Arch [8] and similar.

The repository also allows export and import of already developed components (e.g. developed by third parties). SOFAnodes can be loosely connected together to form *SOFAnet* [26, 23], which provides functionality for automated component trading, licensing, etc.

3.2.2 Deployment docks and deployment

A *deployment dock* can be viewed as a container for launching components and it provides necessary infrastructure for starting, stopping, and updating components.

During deployment of an application, each component forming the applications has to be assigned to a particular dock in the SOFAnode. This assignment is stored in the *deployment plan*, which serves as a "recipe" for launching components. The deployment plan is used by the launching tool, which contacts involved deployment docks and instructs them to launch given components. Necessary code of the components is automatically obtained by the docks from the repository.

3.2.3 Version management at runtime

As described in Sect. 3.2.1, SOFA components are versioned. The versioning model used in SOFA 2.0 is the same as in the original SOFA and it is described in [11]. Due to versioning, the SOFA runtime has to cope with potential class name clashes, which can occur at runtime. By a class name clash, we mean a situation, when two different classes but with the same name have to be loaded into the virtual machine. In Java, this is difficult to do as two different classes or interfaces cannot coexist in the virtual machine unless they are loaded by different classloaders. All potential sources of class name clashes are in detail described in [12].

To solve these class name clashes, SOFA 2.0 uses bytecode manipulation. We successfully used this approach also in the original SOFA implementation, moreover it can be applied in any Java system (for an example see [17]). The approach consist in applying additional postprocessing of Java classes bytecode. During the uploading of classes forming a component into the repository, the bytecode of the classes is modified and the classes are renamed to have unique names. These names are constructed from the original names augmented by the version identifier of the component and thus the uniqueness is reached and all classes can be loaded into the virtual machine by a single classloader.

The approach used is very lightweight and does not introduce any performance and/or usability problems. Moreover, across the other possible and used approaches (also discussed in [12]), our approach is the only one, which can be applied in the scope of the CLDC configuration of Java Micro Edition.

3.3 Controllers

The control logic in SOFA 2.0 is concentrated to dedicated micro-components, which are organized into component aspects. When an application is being launched, the business code of each of its components is weaved with component aspects to equip it with the control logic. The choice and order of the aspects to be applied is defined by an application-specific controller configuration given at deployment time by the deployment plan. To ensure a common minimal functionality, we have specified and implemented a core aspect (and related micro-components) that defines the basic query, binding and life-cycle services.

The definition of micro-components as well as of component aspects are stored in the repository. The weaving of aspects with the business code is performed when a particular component is being instantiated. We do not merge code on the byte-code level, rather we represent each micro-component as a separate Java-object, which makes developing and debugging of micro-components easier. In case of a micro-component that is used to intercept business interfaces, the actual signature of the business interface is not known at the time the micro-component is being developed; thus, the micro-component has to be generated. In our implementation, we use ASM [4] for generating classes for such micro-components at runtime.

When using micro-components to take care of the control logic, it is often necessary to access component's content (i.e. the code provided by a component developer) from micro-components. A typical case is for example to notify the content about a life-cycle change by calling specially annotated methods of the content. In our solution, we pass a reference to the component content to a micro-component during its instantiation.

3.4 Connectors

Software connectors are present in SOFA 2.0 at both design- and runtime. At design time a connector is modeled on a high-level of abstraction using a communication style and properties. At runtime, a connector is present in the form of runtime entities (i.e. Java classes) that realize the prescribed properties. The transition between the high-level semantic description at design time and the implementation classes at runtime is performed by an automatic connector generator [6]. The generation of connectors is done during deployment as one of the steps of creating a deployment plan. The main idea behind postponing the generation to such a late stage of development lifecycle is that at this stage the capabilities of target deployment environment are known (e.g. operating system, installed libraries, network, etc.). By utilizing all this information it is possible to generate highly optimized connector code.

The generated connector code is stored in the repository. The generator also returns descriptors that are needed for connector instantiation. These descriptors are incorporated into a deployment plan to be available at runtime.

Connectors are inherently distributed entities. A single

connector may span different deployment docks. To capture this feature, a connector is divided to a number of *connector units*; a connector unit is a part of the connector that is attached to a component interface and is not distributed any more. The permitted cardinalities and responsibilities of connector units depend on the communication style used. For example in the case of the method invocation communication style, there is one server connector unit and zero or more client connector units.

The communication between a component interface and a connector unit is realized by local method calls. The communication between connector units is typically remote and realized by middleware. The choice of middleware is done by the connector generator and reflects the communication requirements and capabilities of deployment docks. At the local level, connector units are further composed of *connector elements* (Fig. 3), which are typically Java classes that are responsible for implementing particular connector features (e.g. performance monitoring).

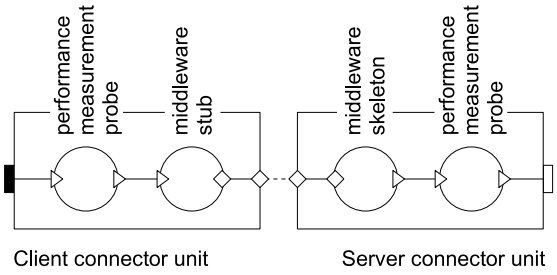

Figure 3. Example of a connector structure

3.4.1 Creating and managing connectors

Connectors are instantiated and managed by connector managers. There is one global connector manager per SOFAnode which keeps track on the relation among connector units, meaning which particular units in the SOFAnode belong to the same connector instance. The global connector manager plays an important role in connecting connector units to form a connector instance. It collects remote references from all connector units belonging to a particular connector and distributes them back so that client connector units may establish connections to server connector units. Apart from a single global connector manager, there are also dock connector managers — one per deployment dock. A dock connector manager is responsible for instantiating a connector unit and registering it with the global connector manager.

To instantiate a connector unit it is necessary to know what implementation to use for it. This information (originally produced by the connector generator) is derived from the deployment plan and fed during application startup to respective dock connector managers in the form of connector unit instantiation templates. Each of these templates associates a triple <*deployment node, component or component instance, interface name*> with a location of the class implementing a particular connector unit. Thus, to instantiate a connector unit at runtime one needs only to know the target component and its interface. This technique makes no difference between connectors instantiated on startup and connectors that may emerge during runtime changes in an architecture (see Sect. 3.5).

3.4.2 Utility interfaces

Connectors are used in SOFA 2.0 not only for mediating communication between components but also to turn a component into a service (e.g. a WebService) and to allow a component to access a service. This is realized by special connector units attached to components' utility interfaces. In the case of a provided utility interface, the connector unit makes it accessible locally and, if desired, it exposes it also externally as a service by registering it in a service registry. In the case of a required utility interface, the attached connector unit allows for local connections or for accessing remote services (e.g. via SOAP). Actually, the only difference compared to connector units attached to "ordinary" component interfaces is in the implementation (i.e. the choice of middleware, capability of registering a service, etc.); however, the way of instantiation (i.e. connector unit instantiation templates) and management of connector units remains the same.

3.5 Runtime changes to an architecture

The controlled evolution of a component application is driven by well-defined evolution patterns. These patterns are supported by the runtime environment which handles reconfigurations according to them. In SOFA 2.0, currently factory pattern and removal pattern are defined.

Technically, factory pattern means creation of a new component and passing a reference to its interface to a caller component that invoked a factory method. This is done in several steps. Once the new component is created, connector units for all its interfaces are created as well. They are instantiated in a standard way by a dock connector manager. (The information about what implementation to use for each connector unit is present in the deployment plan and it was fed to the dock connector manager at startup.) Additionally, it is necessary to create a client connector unit for the interface that is to be returned. The client connector unit (which may be seamlessly used in distributed environment) is used in the rest of the application instead of the original reference to the interface. The client connector

unit is then transmitted through a connector from the factory component to the caller component (if the two component reside in different address spaces, a new client connector unit is created on the caller side), and bound to the required collection interface (i.e. an array of required interfaces), which causes a new client connector unit instance (specific for the caller component) to be created and connected to the same server connector unit as the existing client connector unit has already been connected to. The binding to the collection interface causes new intercepting micro-components associated with the required interface to appear in the control part of the caller component. Such a modification of the control part is performed from a micro-component intercepting the factory method on the caller's side. Finally, a reference to the interface of the newly created component is returned to the calling code; however, to ensure consistency, it is necessary to return a reference to a particular intercepting micro-component associated with the required interface (as opposed to returning a reference directly to the client connector unit).

The removal pattern means destroying a component that was previously created by the factory pattern and removing a reference to it from the caller component's collection required interface.

In addition to the patterns described above, the use of utility interfaces (see Sect. 3.4.2) represents also a kind of architecture evolution, when components' code is responsible for discovering other services and for establishing new bindings to them at runtime.

4 Related work

In this section, we present other contemporary component systems, compare these systems with SOFA 2.0 and discuss their pros and cons. We focus on the component systems, which, like SOFA 2.0, offer support not only for designing components (e.g. just ADL languages) but also provide full component runtime.

Fractal [3] is a component model with very similar capabilities as SOFA 2.0. It offers primitive and composite components, provided and required interfaces, business and control interfaces, etc. The main differences are that 1) Fractal is not defined using a meta-model but rather via a textual specification and a set of interfaces, and that 2) Fractal does not support distribution and multiple communication styles as first-class entities. If one needs connectors and/or different communication styles, then it is necessary to simulate them by components. However, this results in applications where the normal components and these "connector-like" components are mixed and consequently the application's architecture is messy and unclear.

Fractal has a number of implementations. Most complete and relevant with the respect to our work are Julia [3] and AOKell [25]. Julia is a Java-based reference implementation of Fractal, which allows component programming in Java. Components can be created either directly via Julia API or they can be specified using Fractal ADL (an XML-based ADL language); the component implementation then has to follow this specification. Similar to SOFA 2.0, a component is a set of Java classes and interfaces. For implementation of control parts of components, Julia uses so called *mixins*. These are Java classes, which are "mixed" with the original components' classes using bytecode manipulation. Control interfaces and used mixins are specified in Julia configuration file provided at launch time of the application. In the comparison with our microcomponent approach, the Julia's approach is poorly manageable, hard to extend and debug.

Like SOFA 2.0, Julia also supports versioning [16] of components. To avoid potential classname clashes at runtime, Julia incorporates a solution based on a hierarchy of classloaders. The solution works well but in contrary to our technique, it cannot be applied in a platform not featuring Java reflection API, e.g. in the CLDC configuration of Java Micro Edition.

AOKell is also written in Java. Unlike Julia, it has an elaborate mechanism for building control parts of components based on aspect-oriented programming. It separates the implementation of controllers, the content of a component (i.e. business code) and the glue, which ties the controllers to the content. The glue may be realized either with AspectJ [15] or with Spoon [22], which is a compile-time Java processor. The controllers may be componentized, in which case the Fractal ADL is used to describe the structure of a controller. This approach is quite similar to our modeling the control part using microcomponents. Compared to SOFA 2.0, AOKell does not scale well when combining different controllers — for every combination of controllers a new composite component comprising all the controllers has to be defined in Fractal ADL. In SOFA 2.0, it is only necessary to enumerate what controllers shall be used. This advantage is mainly because of the concept of control aspects.

Koala [20] is a component model targeted mainly on embedded devices. It is based on the Darwin component model and offers hierarchical components. Similar to SOFA 2.0, definitions of components and interfaces are stored in a repository but contrary to SOFA 2.0, Koala does not support versioning; once an interface is stored in the repository, it cannot be modified. On the other hand, components themeselves can be modified, but just in a very limited way (new provisions can be added to an existing component, but the existing provisions cannot be removed from it; new requirements can be added, but they must be marked as optional; and similarly, none of the existing requirements can be removed, only they can be marked as optional). In our view,

the SOFA 2.0 solution with complex versioning scheme is more appropriate; a justification can be found in [12]. To support implementation, a Koala compiler is available. It generates C header files from component descriptions. The header files contain definition of types and bindings among components.

ArchJava [1] is another system offering applications built from software components but compared to SOFA 2.0 and Julia it works in a different way. ArchJava is an extension of the Java programing language; it introduces components directly into the language and provides its own compiler, which compiles this enhanced Java into the standard bytecode. The authors claim that their approach of introducing components directly into the language allows tight coupling of an architecture and implementation of component-based applications and prevents inappropriate modification of the architecture at runtime. However, since the component architecture in SOFA 2.0 is managed by the runtime environment according to the model stored in the repository (as opposed to implementing and controlling the architecture by code provided by a developer), no arbitrary changes of the architecture at runtime are possible in SOFA 2.0 either. In addition, the implementation of SOFA 2.0 components is in pure Java and therefore the whole platform can be much more easily integrated with other legacy systems.

ArchJava provides a hierarchical component model like SOFA 2.0 or Fractal. Also, it allows dynamic modifications of architecture at runtime. For creating new component instances, it uses the *new* operator and for new connections, it uses *connection patterns*, which define through which interfaces and to which types of components the new component can be connected. ArchJava does not provide any support for versioning.

Java/A [2] is a component system, which follows the same philosophy as ArchJava — i.e. enhancing the Java language by component constructions. In addition to ArchJava, Java/A provides a behavior specification of components and uses connectors (mainly as adaptors). Compared to SOFA 2.0, Java/A contains the same limitations and issues as ArchJava.

OSGi (Open Services Gateway Initiative) [21] is a platform for deploying and using services in Java, primarily targeted for embedded devices. The basic unit of deployment is a *bundle*, which provides services and which can depend on and use other services. We mention OSGi here because from a higher level of abstraction, a bundle can be seen as a primitive component and its services as interfaces of the component. For versioning of bundles, OSGi employs multiple classloaders and therefore, it cannot be used in devices supporting just CLDC configuration of Java Micro Edition.

Gravity [7] is a component system focusing on dynamic reconfiguration and adaptation of an application. It employs a flat component model, i.e. it provides just primitive components. Gravity is developed upon the OSGi platform and therefore it has the same limitations as OSGi.

5 Conclusion

In this paper, we have presented issues connected with supporting advanced component concepts at runtime and demonstrated our approach on the Java implementation of SOFA 2.0 component system. Specifically, we have focused on the implementation of advanced features like support for transparent distribution using connectors, coexistence of multiple versions of the same component at runtime, runtime evolution of an application architecture, and control part of components.

At present, the described parts of the SOFAnode are either finished and fully functional or they are close to their completion. Also, we are working on a graphical development environment.

Acknowledgments

This work was partially supported by the Czech Academy of Sciences project 1ET400300504 and partially supported by the ITEA/EUREKA project OSIRIS Σ!2023.

References

[1] Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting Software Architecture to Implementation, Proceedings of ICSE 2002, Orlando, USA, May 2002

[2] Baumeister, H., Hacklinger, F., Hennicker, R., Knapp, A., Wirsing, M.: A Component Model for Architectural Programming, In Electronic Notes in Theoretical Computer Science, Vol. 160, Aug 2006

[3] Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: The Fractal Component Model and Its Support in Java, Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 36(11-12), 2006

[4] Bruneton, E., Lenglet, R., Coupaye, T.: ASM: A code manipulation tool to implement adaptable systems, `http://asm.objectweb.org/`

[5] Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, Proceedings of SERA 2006, Seattle, USA, IEEE CS, Aug 2006

[6] Bures, T.: Generating Connectors for Homogeneous and Heterogeneous Deployment, Ph.D. Thesis, Department of Software Engineering, Mathematical and Physical Faculty, Charles University, Prague, Sep 2006

[7] Cervantes, H., Hall, R. S.: A Framework for Constructing Adaptive Component-based Applications: Concepts and Experiences, Proceedings of CBSE 2004, Edinburgh, Scotland, May 2004

[8] GNU Arch, http://www.gnu.org/software/gnu-arch/

[9] Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, Third Edition, http://java.sun.com/docs/books/jls

[10] Hnetynka, P., Plasil, F.: Dynamic Reconfiguration and Access to Services in Hierarchical Component Models, Proceedings of CBSE 2006, Vasteras, Sweden, LNCS 4063, Jun 2006

[11] Hnetynka, P., Plasil, F.: Distributed Versioning Model for MOF, Proceedings of WISICT 2004, Cancun, Mexico, Jan 2004

[12] Hnetynka, P., Tuma, P.: Fighting Class Name Clashes in Java Component Systems, Proceedings of JMLC 2003, Klagenfurt, Austria, Aug 2003

[13] Eclipse Modeling Framwork, http://www.eclipse.org/emf/

[14] Enterprise Java Beans specification, version 2.1, Sun Microsystems, Nov 2003

[15] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ, In Proceedings of ECOOP 2001, June 18-22, 2001, Budapest, Hungary, LNCS 2072, Springer, 2001

[16] Kornas, J., et al: Support pour la reconfiguration d'implantation dans les applications a composants Java, DECOR04, Grenoble, France, Oct 2004

[17] Luer, C., van der Hoek, A.: JPloy: User-Centric Deployment Support in a Component Platform, Proceedings of CD 2004, Edinburgh, UK, May 2004

[18] Mencl, V., Bures, T.: Microcomponent-Based Component Controllers: A Foundation for Component Aspects, Proceedings of APSEC 2005, Taipei, Taiwan, Dec 2005

[19] OMG: CORBA Components, v 3.0, OMG document formal/02-06-65, Jun 2002

[20] van Ommering, R., van der Linden, F., Kramer, J., Magee, J., The Koala Component Model for Consumer Electronics Software, In IEEE Computer, Vol. 33, No. 3, pp. 78-85, Mar 2000

[21] Open Services Gateway Initiative, http://www.osgi.org/

[22] Pawlak, R: Spoon: Compile-time Annotation Processing for Middleware, IEEE Distributed Systems Online, vol. 7, no. 11, 2006, art. no. 0611-oy001

[23] Plasil, F., Balek, D., Janecek, R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998

[24] Rouvoy, R., Merle, P.: Leveraging Component-Oriented Programming with Attribute-Oriented Programming, In Proccedings of WCOP 2006, Nantes, France, July 2006

[25] Seinturier, L., Pessemier, N., Duchien, L., Coupaye, T.: A Component Model Engineered with Components and Aspects, CBSE'06, LNCS 4063, Jun 2006

[26] Sobr, L., Tuma, P.: SOFAnet: Middleware for Software Distribution over Internet, Proceedings of SAINT 2005, Trento, Italy, Feb 2005

[27] Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edition, Addison-Wesley, Jan 2002