Distributed Component System Based On Architecture Description: The SOFA Experience

Tomáš Kalibera and Petr Tůma

Charles University
Faculty of Mathematics and Physics,
Department of Software Engineering
Malostranské náměstí 25, 118 00 Prague 1,
Czech Republic
`kalibera@nenya.ms.mff.cuni.cz` `petr.tuma@mff.cuni.cz`

Abstract. In this paper, the authors share their experience gathered during the design and implementation of a runtime environment for the SOFA component system. The authors focus on the issues of mapping the SOFA component definition language into the C++ language and the integration of a CORBA middleware into the SOFA component system, aiming to support transparently distributed applications in a real-life environment. The experience highlights general problems related to the type system of architecture description languages and middleware implementations, the mapping of the type system into the implementation language, and the support for dynamic changes of the application architecture.

Keywords. Architecture description languages, ADL, component definition languages, CDL, middleware, CORBA, language mapping, dynamic architectures.

1 Introduction

The notion of components enjoys significant interest in the software engineering community. Components are considered to be useful *units of code sharing and reuse*, as well as useful *building blocks of software architectures*. While the former view is supported by practical component systems [15, 19, 24], the latter view appears to be lagging behind. The current trend of modeling software architectures using UML is criticized as being inadequate [8], and while the research in component systems based on architecture description languages (ADL component systems) cites inarguable benefits of such systems [1, 5, 13, 14, 23], the described projects rarely get past research prototypes.

The discrepancy between the cited benefits of ADL component systems and the lack of their practical employment leads us to believe that there are unresolved issues that prevent this employment. In order to investigate these issues, we have designed and implemented a runtime environment for the SOFA ADL component system [21] (SOFA environment).

Our chief goal in the design and implementation of the SOFA environment is to support development of transparently distributed applications. The development centers around a hierarchical description of the application architecture. This description is gradually refined from a coarse granularity level, where components correspond to implementation modules, to a fine granularity level, where components correspond to implementation objects. These components are then mapped to implementation objects using a standardized language mapping, with the architecture description defining the interconnection of these objects into the component application. When the application is run, its components can be deployed onto several network hosts. The components that share a host are interconnected through linking and run in one address space. The components that run on different hosts are interconnected through connectors.

The SOFA environment describes the application architecture using the SOFA component definition language [10, 21] (SOFA CDL). SOFA CDL is mapped into C++, which is used to implement the components. The connectors are built using CORBA [18]. The SOFA environment also allows interfacing the application with GNOME [26] to provide user interface support. The choice of GNOME as a representative of a component framework and CORBA as a representative of an off-the-shelf middleware allows us to evaluate how the SOFA environment supports real-life applications in a real-life environment.[1]

The paper continues by a brief introduction of the SOFA component model and SOFA CDL in Sect. 2. The description of the design and implementation of the SOFA environment follows in Sect. 3. Our experience with the CDL to C++ mapping and the integration of CORBA, as well as the ability of the SOFA environment to support applications, is evaluated and generalized for a broad class of ADL component systems in Sect. 4. Section 5 relates this paper to other work in the field of ADL component systems. The paper is concluded in Sect. 6.

2 SOFA Component Model And SOFA CDL

The SOFA component model [21] views an application as a hierarchy of nested software components. A component is an instance of a component *template*, which consists of a component *frame* and a component *architecture*. The frame lists all interfaces that the component *requires* and *provides*. The architecture implements the operations of the provided interfaces, relying only on the operations of the required interfaces. A frame can be implemented by several architectures.

An architecture is either *composed* or *primitive*. A composed architecture defines a *composed component* as built from *subcomponents* by listing the frames of the subcomponents and the *ties* between the interfaces of the component and the subcomponents. A primitive architecture defines a *primitive component* as implemented in an implementation language outside the scope of the component model.

[1] The SOFA ADL component system also includes a Forte IDE and a Java runtime. These are outside the scope of this paper.

A tie between the interfaces of a component and its subcomponents can be of three types. *Binding* denotes connecting a required interface of a subcomponent to a provided interface of a subcomponent. *Delegating* denotes connecting a provided interface of a component to a provided interface of its subcomponent. *Subsuming* denotes connecting a required interface of a subcomponent to a required interface of its component.

An example of the interface, frame and architecture definitions in SOFA CDL is in Fig. 1. The example defines a variation of the ubiquitous "Hello World" application that prints a greeting. The application is an instance of a component with the `ApplicationArch` architecture, which implements the `ApplicationFrame` frame. The `Message` subcomponent provides the greeting to be displayed, the `Display` subcomponent provides the functionality to display a message, the `HelloWorld` subcomponent uses the two other subcomponents to display the greeting. The application defined by the example will be used in other examples throughout the paper.

```
interface MessageIface { string message (); };
interface DisplayIface { void print (in string message); };

frame MessageFrame { provides: MessageIface MessageProv; };
frame DisplayFrame { provides: DisplayIface DisplayProv; };

frame HelloWorldFrame {
  requires: MessageIface MessageReq; DisplayIface DisplayReq;
  provides: ApplicationIface ApplicationProv;
};

architecture MessageArch implements MessageFrame primitive;
architecture DisplayArch implements DisplayFrame primitive;
architecture HelloWorldArch implements HelloWorldFrame primitive;

architecture ApplicationArch implements ApplicationFrame {
  inst MessageFrame Message;
  inst DisplayFrame Display;
  inst HelloWorldFrame HelloWorld;
  bind HelloWorld:MessageReq to Message:MessageProv;
  bind HelloWorld:DisplayReq to Display:DisplayProv;
  delegate ApplicationProv to HelloWorld:ApplicationProv;
};
```

Fig. 1. A SOFA CDL definition of an application architecture.

SOFA CDL can also specify semantics of interfaces and frames using behavior protocols [22] and employ complex connectors [3, 4]. These are outside the scope of this paper.

3 SOFA Environment

The SOFA environment defines and implements a mapping of SOFA CDL into C++ used to map components to implementation objects, implements a deployment mechanism used to deploy the components onto network hosts and to interconnect the components, and implements the connector generator used to produce connectors between components that run on different hosts. These three parts of the SOFA environment are described in this section.

3.1 Mapping SOFA CDL Into C++

The CDL to C++ mapping is based on the IDL to C++ mapping of CORBA [16]. Similar to CORBA IDL, the type system of SOFA CDL is independent of the implementation languages of components and has a standardized mapping into these languages. Making the type system independent on the implementation language makes it easier to generate connectors and potentially also to support multiple implementation languages of components.

The mapping of the types that SOFA CDL shares with CORBA IDL follows the IDL to C++ mapping. The types original to SOFA CDL, namely frames and architectures, are mapped into the frame and architecture classes that follow the approach used to map interfaces with attributes.

A frame class has accessor methods for the provided and required interfaces of the frame, which are represented as protected references to the classes that map the interfaces. An example of a generated frame class is in Fig. 2. To allow substitution of components with the same frame but different architectures, the frame class is a virtual base class that is inherited by architecture classes of the architectures implementing the frame.

```
class HelloWorldFrame : virtual public FrameBase {
  public:
    // Accessor methods generated for provided and required interfaces
    inline virtual ApplicationIface_ptr ApplicationProv () {
      return (ApplicationIface::_duplicate (pApplicationProv)); };
    inline virtual void ApplicationProv (const ApplicationIface_ptr value) {
      pApplicationProv = ApplicationIface::_duplicate (value); };
  protected:
    ApplicationIface_ptr pApplicationProv;
    . . .
};
```

Fig. 2. A generated C++ mapping of HelloWorldFrame.

The implementation of an architecture class differs for composed and primitive architectures. An architecture class of a composed architecture has accessor methods for the subcomponents of the architecture, which are represented as private references to the frame classes of the frames of the subcomponents. The architecture class also contains code that allows to set up the ties between interfaces as defined by the `bind`, `delegate` and `subsume` clauses in the architecture definition.

For performance reasons, the code does not interconnect the interfaces of the composed component with the interfaces of its subcomponents directly. Instead, it allows propagating references to the provided interfaces of primitive components along the ties of the architecture definition by the `createBindingsAndDelegates` and `createSubsumes` methods. The required interfaces of primitive components are thus tied directly to the provided interfaces, with the composed components whose boundaries the ties cross adding no overhead to the invocations of methods accessible through these ties.

An example of a generated composed architecture class is on Fig. 3.

```
class ApplicationArch :
  virtual public ApplicationFrame, virtual public ArchitectureBase
{
  public:
    // Methods generated for setting up the ties between interfaces
    virtual void createBindingsAndDelegates () {
      iHelloWorld->MessageReq (iMessage->MessageProv ());
      iHelloWorld->DisplayReq (iDisplay->DisplayProv ());
      pApplicationProv = iHelloWorld->ApplicationProv (); };
    virtual void createSubsumes () {
      iMessage->createSubsumes ();
      iDisplay->createSubsumes ();
      iHelloWorld->createSubsumes (); };
  private:
    HelloWorldFrame_ptr iHelloWorld;
    ...
};
```

Fig. 3. A generated C++ mapping of ApplicationArch.

An architecture class of a primitive architecture is a virtual base class that the implementation of the primitive component inherits from. An example of an implementation of a primitive component is on Fig. 4. The example uses nested classes to implement the provided interfaces, and demonstrates how both the provided and the required interfaces of a frame are accessed by the implementation of the primitive component.

The frame and architecture classes also inherit from base classes that define methods for generic access to the provided and required interfaces of the frame and the subcomponents and the ties of the architecture. These methods are required by the deployment mechanism.

```
class HelloWorld : public virtual HelloWorldArch {
  public:
    // Implementation of the ApplicationIface interface
    class Application : public virtual ApplicationIface {
      public:
        Application (HelloWorld *frame) { me = frame; };
        // Displaying the greeting using the other subcomponents
        virtual Short run (const StringSequence& args) {
          char *message = me->MessageReq()->message ();
          me->DisplayReq()->print (message);
          return 0;
        };
      private:
        HelloWorld *me;
    };
    // Initialization of the HelloWorldArch architecture
    virtual void initialize () {
      HelloWorldArch::initialize ();
      ApplicationProv (new Application (this));
    };
};
```

Fig. 4. A C++ implementation of HelloWorldArch.

3.2 Deploying Application Components

The deployment is configured by a deployment descriptor. For each frame, the deployment descriptor specifies the architecture that the component will use and the host where the component will run. The deployment is controlled from a single place and expects each host to run a simple server that allows remote instantiation of components. The initialization and interconnection methods of the component are then invoked remotely on the component itself.

The control flow of the deployment mechanism follows the hierarchical architecture of the application being deployed. The architecture forms a tree with each node representing a component. Nodes representing composed components are parents of nodes representing their subcomponents. Nodes representing primitive components are leaves. The references to provided and required interfaces are attributes of each node.

At the beginning of the deployment process, the references to provided interfaces are stored in the attributes of nodes representing primitive components. The references are then propagated toward the root of the tree along the bind and delegate ties in one tree traversal pass, and toward the leaves of the tree along the subsume ties in another traversal pass. The process uses the `createBindingsAndDelegates` and `createSubsumes` methods defined by the language mapping of the component architectures.

```
typedef sequence<string> StringSequence;
interface ApplicationIface { short run (in StringSequence args); };
frame ApplicationFrame { provides: ApplicationIface ApplicationProv; };
```

Fig. 5. The application frame.

The deployment expects the application to implement a standardized frame in Fig. 5. After the application is deployed, the `run` method of `Application-Iface` provided by the application is invoked to launch the application.

3.3 Generating Connectors Using CORBA

Connectors are used to interconnect components that run on different network hosts by delivering remote method invocations to the components. As the hosts where components should run are only known at deployment time, the connectors have to be generated and dynamically loaded at deployment time.

Although the SOFA environment does not place any principal restrictions on the middleware used to implement connectors, we have focused on connectors that are generated by off-the-shelf CORBA middleware. The connector generator is flexible enough to support a number of CORBA middleware implementations.

CORBA middleware generates connectors from a CORBA IDL definition of the interfaces that the connector delivers invocations to. An IDL compiler accepts the CORBA IDL definition of an interface as input and generates C++ source code of the stub and skeleton parts of the connector as output. Both parts need to be compiled, the stub part of the connector is then called by the components that require the interface, the skeleton part of the connector then calls the components that provide the interface.

A development environment that includes both an IDL compiler and a C++ compiler is needed to generate a connector. To avoid the need of having this environment available at deployment time, the SOFA environment pregenerates a set of connectors for all interfaces of an application.

For each interface, a CORBA IDL file that contains the definition of the interface and includes the definitions of all types that the interface relies on is generated by the SOFA environment. The file is compiled by the IDL compiler to yield the C++ source code of the stub and skeleton parts of the connector. The SOFA environment also generates C++ source code of the connectors that uses the code generated by the IDL compiler and interfaces it with the components. The C++ source code is compiled into a pregenerated connector. At deployment time, the pregenerated connectors are dynamically linked with the components.

The SOFA environment can be configured at deployment time to use several middleware implementations. All connectors of a single middleware implementation are managed by a single connector manager. The task of the connector manager is to provide access to the listening loop of the middleware and to enable creation of stub and skeleton parts of a connector in a middleware independent manner.

```
class ConnectorManager {
  public:
    virtual ObjectBase_ptr loadStubPart (const char *reference) = 0;
    virtual char *loadSkeletonPart (ObjectBase_ptr servant) = 0;
    virtual void startListening () = 0;
    virtual void stopListening () = 0;
};
```

Fig. 6. The connector manager interface.

The interface of the connector manager is in Fig. 6. The `loadSkeletonPart` method creates the skeleton part of a connector, returning a stringified reference of the target interface. The `loadStubPart` method creates the stub part of a connector, accepting this stringified reference. The `startListening` and `stopListening` methods control the listening loop of the middleware.

4 Experience in Retrospective

4.1 Shareable Language Mapping

The initially most visible feature of the SOFA environment was the CDL to C++ mapping, based on the IDL to C++ mapping of CORBA [16]. The CDL to C++ mapping of the data and interface types, which SOFA CDL shares with CORBA IDL, is almost as complex as the IDL to C++ mapping of these types. In addition to the data and interface types, the CDL to C++ mapping also supports the frame and architecture types. Considering the size of the IDL to C++ language mapping, over 170 pages of specification at this time, the CDL to C++ language mapping is obviously far from trivial.

The complexity of the mapping can introduce extra cost in terms of code size and runtime overhead. In principle, the extra cost of a mapping designed solely for use by the component code does not have to exceed the extra cost introduced by other libraries that provide useful types in the C++ environment, such as STL [11]. The problem particular to the CDL to C++ mapping, and a language mapping used by any other component system that aspires to employ off-the-shelf middleware to build connectors, is that the mapping is used both by the component system and by the middleware. A typical situation in this case is that the mapping used by the component system is not compatible with the mapping used by the middleware, prompting the need for deep copying at best, and deep copying and data conversion at worst, of all data passed through the middleware. Given the performance of contemporary middleware implementations [25], the copying and conversion might be acceptable in an explicitly distributed application that employs the middleware in a few carefully selected points, but not in a transparently distributed application that relies on the middleware for interconnecting its components at fine granularity levels, where components correspond to implementation objects.

The problem of compatibility of the language mappings used by the component system and the middleware implementations employed to build connectors can be solved by sharing the mapping among the component system and the middleware implementations. In most cases, this requires extending the language mappings of contemporary middleware implementations.

A language mapping of a contemporary middleware implementation is typically designed to make it possible to write applications that are portable across middleware implementations. The mapping is defined so that the application employing the middleware can easily access the mapped types, but it does not define how the middleware itself accesses the mapped types.

A language mapping that is to be shared among a component system and middleware implementations has to extend the contemporary mappings by defining how the middleware itself accesses the mapped types. Such a language mapping makes it possible not only to write applications that are portable across middleware implementations, but also to write middleware implementations that can share language mappings and thus coexist in a single application without incurring extra cost in terms of code size and runtime overhead. A step in this direction are the ORB portability interfaces in the IDL to Java mapping of CORBA [17].

4.2 Connectors Built Using CORBA

The separation of development and deployment phases of the application lifecycle implies a need to postpone the decision on what connectors to employ from the development time to the deployment time. This goes contrary to the typical usage of off-the-shelf middleware, where the connectors are generated and integrated into the application at the development time.

Although it is theoretically possible to use off-the-shelf middleware to generate connectors at deployment time, such an approach runs into a number of practical difficulties. First, it is unusual to require the development system of the middleware to be available at deployment time. Second, the development system of the middleware is often interactive and thus hard to integrate into the component system.

Alternatively, a set of connectors for all interfaces of an application can be pregenerated at the development time. Only those pregenerated connectors that are actually employed will be used at the deployment time. Our experience demonstrates that while feasible, this approach runs against the typical usage of off-the-shelf middleware, where connectors for multiple interfaces are generated from a single CORBA IDL file.

When connectors for multiple interfaces are generated from a single CORBA IDL file, the middleware produces a monolithic module that contains the marshalling code together with the mapping of all types used by the connectors. When used to generate the connectors for one interface at a time, the middleware produces modules that are largely redundant in mapping of those types that are shared by the connectors. Even though the redundancy can be removed during function level linking, the time spent generating and compiling redundant code is prohibitive even for relatively small number of types and interfaces.

To avoid the problems of redundancy when employed in a component system, an off-the-shelf middleware should provide features that allow for separated generation of the marshalling code and the mapping of the types used by the connectors. Provided that the mapping of the types could be shared among the component system and the middleware implementations, this would allow for generating the mapping of the types at development time, and generating the marshalling code on demand at deployment time.

4.3 ADL Type System Not Suitable

In retrospect, the most constraining decision with respect to the usability of the component system was basing the SOFA type system on the CORBA type system. The type system of CORBA is tailored to suit the underlying remote procedure call mechanism, which is acceptable because a CORBA application uses IDL interfaces in a few carefully selected points. When carried over to SOFA, the type system becomes much more restrictive because a SOFA application uses CDL interfaces for interconnecting its components at fine granularity levels, where components correspond to implementation objects.

Looking at the differences between the type system of C++, which is normally used in the environment we consider, and the type system of SOFA, we can see that C++ relies heavily on reference and pointer types that may not have a counterpart in the SOFA type system.

Reference and pointer types that are used to pass data by reference, whether merely for sake of efficiency or to allow modification of the data, have a good match in the SOFA types used to pass the same data in one of the in, out or inout directions.

Reference and pointer types that are used to build dynamic data structures do not have a good match in the SOFA types. Even if the dynamic data structure happens to match the SOFA sequence or value types, the sharing semantics applied by C++ will not match the copy semantics applied by SOFA. The sharing semantics of the reference and pointer types is difficult to mimic in a component system that supports transparently distributed applications. When building dynamic data structures, it is therefore better to employ high level tools such as containers and iterators rather than low level tools such as references and pointers.

A component system can provide containers and iterators modeled after STL [11] or another well tested framework. These can be employed to build dynamic data structures without having to associate a specific sharing or copy semantics with the type, which would be difficult to implement when the type is used both by C++ and by SOFA.

Reference and pointer types that are used to denote objects may appear to have a good match in the SOFA object reference type. Instances of both types give their holder the ability to invoke methods on an object. Implementation of a component system that employs this similarity is possible, although not without difficulties [3]. Reference and pointer types that are used to denote functions represent a similar case.

4.4 Need Anticipated Dynamic Changes

From the architectural point of view, passing references that denote objects has the effect of creating new ties between components. Together with the ability to instantiate components, this provides a mechanism for dynamically changing the architecture of the application. The mechanism is similar to the one normally employed by object oriented applications to introduce dynamic changes by creating and linking objects. This similarity makes it well suited for supporting anticipated dynamic changes of the architecture of component applications. The flexibility and ease of use of the mechanism supersedes that of many contemporary component systems with architecture description languages [5, 14].

The downside of the mechanism is that the new connections and components are not reflected in the architecture description. This makes the architecture description lose its relevancy to the application architecture it is to describe. This problem exists in most component systems that employ architecture description languages, where the architecture description is either static [5, 23], or expressed in a way that does not lend itself to describing anticipated dynamic changes [2, 12].

Anticipated dynamic changes of the application architecture appear to be of fundamental importance, much more so than the unanticipated dynamic changes the software architecture research community focuses on. If the architecture description is to be used in a component system at fine granularity level, it is necessary to extend the architecture description language to support such changes. Following the approach suggested for building dynamic data structures, the dynamic architectures could be described as dynamic collections of components.

4.5 Legacy Components And Connectors

Integrating the component system with CORBA and GNOME gave rise to the need of supporting legacy components, especially the components of GNOME used to build the user interface. Besides running into problems with the type system outlined earlier, we also encountered problems related to legacy distribution mechanisms.

The graphical user environment of GNOME runs on top of the X Window System [20], which relies on its own distribution mechanism. The legacy components of GNOME use X resource identifiers as references. The distribution mechanism of the X Window System should therefore be regarded a middleware and X protocol connectors should be introduced to interconnect X components. This would have the advantage of using the X protocol, which is more efficient than the protocols of general purpose middleware. More work needs to be done to design a mechanism for cooperation between multiple types of middleware.

5 Related Work

Although a number of ADL component systems exists, most share the basic architectural concepts related to components and connectors. The component model of SOFA is no exception, being similar to the component model of Darwin [13]. It also fits well into the ACME framework [9] and the xADL toolkit [7], which provide a basis for sharing and manipulating architectural information.

What distinguishes our work on SOFA from that carried out on other component systems is the close integration of our SOFA implementation with CORBA and GNOME. To our knowledge, few other ADL component systems come close to this level of implementation. The notable exceptions are the C2 [14] and Rapide [12] projects, both exerting effort to support real-life applications in real-life settings. Neither project, however, aims at supporting transparently distributed applications.

With its design and implementation, the SOFA environment is also close to component systems that are not based on formal architecture description, such as Microsoft COM [15] or Sun EJB [24]. Besides the lack of the architecture description itself, these systems differ from the SOFA environment also by omitting the explicit specification of interfaces required by components, which is needed for rigorous assembly of components.

Although also lacking the formal architecture description, more similar to the SOFA environment is the CORBA Component Model [19], which provides a definition of components with explicit specification of provided and required interfaces. We believe that the need for shareable language mapping, identified in this paper, also concerns the CORBA Component Model.

Also related to our work is the development in middleware implementations, especially in the area of reflective middleware. Reflective middleware implementations are generally more modular and thus lend themselves better to integration with a component system. Reflective middleware can also employ the formal architecture description for its configuration [6].

6 Conclusion

We have presented the design and implementation of a runtime environment for the SOFA component system. The implementation is integrated with GNOME and CORBA as representatives of a contemporary component framework and a distributed middleware.

The SOFA environment features a CDL to C++ mapping, a deployment mechanism and a connector generator. The language mapping is easy to use, introduces little overhead per se, and enables component substitution. The deployment mechanism is configurable and supports both single-host and distributed deployment transparent to the application. The connector generator produces connectors independent of the application and can integrate several CORBA middleware implementations.

The SOFA environment meets our goals of supporting real-life applications in real-life settings, vital to discover the limitations of ADL component systems with respect to applications. The paper further highlights our findings in this respect, related to the type system of architecture description languages and middleware implementations, the mapping of the type system into the implementation language, and the support for dynamic changes of the application architecture.

We argue that the type system of the architecture description languages needs to be enriched to support building of dynamic data structures without having to resort to the low level tools such as references and pointers, which do not lend themselves well to transparent distribution.

We point out that the mapping of the type system used by contemporary off-the-shelf middleware needs to be extended to define those features of the mapped types that the implementations of the middleware rely upon. This allows sharing the language mapping among the component system and the middleware implementations used to build the connectors. For efficiency reasons, the middleware should generate the marshalling code and the mapping of the types separately.

We also emphasize that the dynamic changes of the application architecture should be allowed through a mechanism similar to the one normally employed by applications to introduce dynamism, such as creating and linking objects. The architecture description languages should reflect this mechanism and provide support for anticipated dynamic changes.

We believe that our findings are not constrained to the particular design and implementation of the SOFA environment we have described, but can be generalized to cover the broad class of component systems that employ architecture description languages or other forms of formal architecture description.

The implementation of the SOFA environment is available for download at http://nenya.ms.mff.cuni.cz.

Acknowledgments

The authors would like to thank František Plášil, Stanislav Višňovský and Adam Buble for valuable comments, and all the members of the Distributed Systems Research Group at Charles University for their work on the SOFA project.

References

1. Allen R. J.: A Formal Approach to Software Architecture, Doctoral thesis at Carnegie Mellon University, USA, 1997
2. Allen R. J., Douence R., Garlan D.: Specifying and Analyzing Dynamic Software Architectures, Proceedings of FASE 1998, Portugal, 1998
3. Bálek D.: Connectors in Software Architectures, Doctoral thesis at Charles University, Czech Republic, http://nenya.ms.mff.cuni.cz, 2002
4. Bálek D., Plášil F.: Software Connectors and Their Role in Component Deployment, Proceedings of DAIS 2001, Poland, 2001

5. Bellissard L., Ben Atallah S., Boyer F., Riveill M.: Distributed Application Configuration, Proceedings of ICDCS 1996, Hong Kong, 1996
6. Blair G., Blair L., Issarny V., Tůma P., Zarras A.: The Role of Software Architecture in Constraining Adaptation in Component-based Middleware Platforms, Proceedings of Middleware 2000, USA, 2000
7. Dashofy E. M., van der Hoek A., Taylor R. N.: An Infrastructure for the Rapid Development of XML-based Architecture Description Languages, Proceedings of ICSE 2002, USA, 2002
8. Garlan D., Kompanek A.: Reconciling the Needs of Architectural Description with Object-Modeling Notations, Proceedings of UML 2000, United Kingdom, 2000
9. Garlan D., Monroe R., Wile D.: ACME: An Architecture Description Interchange Language, Proceedings of CASCON 1997, Canada, 1997
10. Hnětynka P., Mencl V.: Managing Evolution of Component Specifications using a Federation of Repositories, Technical report 2001/2, Department of Software Engineering, Charles University, Czech Republic, 2001
11. International Organization for Standardization: C++ Programming Language, ISO/IEC standard 14882, 1998
12. Luckham D. C., Kenney J. J., Augustin L. M., Vera J., Bryan D., Mann W.: Specification and Analysis of System Architecture Using Rapide, IEEE Transactions on Software Engineering 21(4), 1995
13. Magee J., Tseng A., Kramer J.: Composing Distributed Objects in CORBA, Proceedings of ISADS 1997, Germany, 1997
14. Medvidovic N., Taylor R. N., Whitehead E. J.: Formal Modeling of Software Architectures at Multiple Levels of Abstraction, Proceedings of CSS 1996, USA, 1996
15. Microsoft: Component Object Model Specification 0.9, http://www.microsoft.com, 1995
16. Object Management Group: C++ Language Mapping Specification, formal/99-07-41, ftp://ftp.omg.org/pub/docs/formal/99-07-41.pdf, 1999
17. Object Management Group: Java Language Mapping Specification, formal/99-07-53, ftp://ftp.omg.org/pub/docs/formal/99-07-53.pdf, 1999
18. Object Management Group: Common Object Request Broker: Architecture and Specification, CORBA 2.6.1, formal/02-05-08, ftp://ftp.omg.org/pub/docs/formal/02-05-08.pdf, 2002
19. Object Management Group: CORBA Component Model Specification, ptc/01-11-03, ftp://ftp.omg.org/pub/docs/ptc/01-11-03.pdf, 2001
20. Open Group: X Windows System, http://www.x.org, 2002
21. Plášil F., Bálek D., Janeček R.: SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, Proceedings of ICCDS 1998, USA, 1998
22. Plášil F., Višňovský S.: Behavior Protocols for Software Components, IEEE Transactions on Software Engineering 28(9), 2002
23. Shaw M., DeLine R., Klein D. V., Ross T. L., Young D. M., Zelesnik G.: Abstractions for Software Architecture and Tools to Support Them, IEEE Transactions on Software Engineering 21(4), 1995
24. Sun Microsystems: Enterprise JavaBeans Specification 2.0, http://www.microsoft.com, 2002
25. Tůma P., Buble A.: Open CORBA Benchmarking, Proceedings of SPECTS 2001, USA, 2001.
26. GNOME Documentation Project, http://developer.gnome.org/projects/gdp, 2002