Update description language
(position paper)

Vladimír Mencl[1], Zuzana Petrová[1], František Plášil[1,2]
{mencl,petrova,plasil}@nenya.ms.mff.cuni.cz

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

[2]Institute of Computer Science
Academy of Sciences of the Czech Republic
Pod vodárenskou věží 2, 180 00 Prague 8
Czech Republic

23.6.1999

Abstract

In order to allow for component evolution and dynamic updating, it is necessary to develop facilities for describing differences between versions of components, and for a (possibly automatic) adapting mechanisms in cases of "slight" incompatibility. This paper proposes an *Update description language (UDL)*, designed for describing changes made both in a component interface and in component implementation. Together with the *Component implementation description language*, UDL allows to express transition of the state of a component's old to its new version at a runtime update.

1 Introduction

This paper presents the current status of the research on describing changes in software components in the SOFA/DCUP environment. SOFA (**SOF**tware **A**ppliances) is a project run by *The CORBA and Distributed Systems Research Group* (`http://nenya.ms.mff.cuni.cz/thegroup`) ([PBJ97], [PBJ98a], [PBV99a]). DCUP is an architecture focused on **D**ynamic **C**omponent **UP**dating in the SOFA environment. This paper is a position paper which reports on the current status of the SOFA/DCUP research with respect to description of component updates.

2 Issues and challenges

This section shall describe the challenges associated with the task of providing facilities for handling component updates. The challenges are component externalization, and maintaining component updates — this includes handling changes made to either to the interface of a component, or to a component's implementation.

The following subsections present issues related to these challenges, the goals of the paper are pointed out at the end of this section.

2.1 Component externalization

2.1.1 Describing component implementation

In [CDL98], the component definition language is specified. However, this language focuses on the architectural aspects of the components only. For the purposes of dynamic updating, it is necessary to have an opportunity to describe the implementation of a primitive component. The main purpose of this description is to enable externalizing the state of the component, but this description can also be used to generate a skeleton of the component initialization code.

2.1.2 Storing and restoring state of a component

In a dynamically evolving architecture, it is necessary to have an opportunity to manipulate the state of a component. The following two issues requiring component externalization are obvious, however, other may arise:

1. Component migration

   It should be possible in SOFA to transparently migrate a component from one node to another. In order to accomplish that, it is necessary to be able to externalize and transfer the component's state.

2. Dynamic component updating

   In order to update a stateful component in DCUP, it is necessary to transfer the state from the old version of the component instance to the new one. One possible way of doing so is by externalizing the state of the component into a persistent store, possibly transforming the state from the format of the old version into the format of the new version if these differ, and then restoring the state in the new version.

2.2 Needs for evolving software

In an environment considering only static configuration, it would be sufficient to have a static description of software components capturing their state at the time they were developed, and to use these descriptions to assembly an application.

But in an evolving environment, new versions of software components arise, and from time to time, it is even necessary to modify the interfaces of the component.

For various reasons, which will be explained below, it is necessary to allow both the old and new version of a type coexist in a common namespace. To allow for this, it is necessary for these different versions of a type to have a unique identification, although the developer may wish to refer to both of them by a shared common name.

2.2.1 Entities expected to be subject to a change

In the SOFA/DCUP model, the following types of changes are expected:

1. changes of component implementation

2. modifications of types

Ad 1: Focus is put on the transition of the externalized state of the component. For situations when the format of the externalized data changes, it is necessary to provide facilities for expressing the difference in format and transformation rules to handle these differences, and also to provide runtime facilities executing these rules. This paper partially shows how we have dealt with this challenge.

Ad 2: For changes made to *component border elements* (which consist of interfaces, types derived from interfaces and/or used to derive interfaces, and also of component frames), it is necessary to provide a framework which would, among others, allow coexistence of components using different versions of component border elements in a single SOFAnode, in a single application, even in a single composed component, and therefore even in a single type namespace of a target implementation language[1].

This coexistence is necessary for bindings between different versions of a type. We should face the challenge of allowing such bindings, although it is not possible to find an ideal solution. However, it is necessary to allow such bindings to avoid replacing the whole application when an interface of a subcomponent changes. In section 4, the solution we've chosen shall be presented.

[1]for the *proof of the concept* implementation, we have chosen *java* as the implementation language

2.2.2 Descriptions of new versions and/or descriptions of differences between versions

To handle the goals mentioned in the previous subsection, it is necessary for the component development site to contain description of both the old and new version (of all of the component border elements).

But it is also necessary to have a description of the differences between these two versions. Such a description would be used to obtain information about compatibility of these versions, and possibly could serve as a base for automatical creation (generating) of an *adaptor* in case of *partial incompatibility.*[2].

It is easy to see, that these three descriptions together contain some redundancy. Either the description of the new version could be derived from the description of the old version and the description of the differences, or vice-versa, the description of the differences could be obtained by comparing descriptions of the two respective versions.

However, not all information, which should be contained in the automatically generated description, can be obtained from the two others, some information has to be provided by the developer. It is a goal of this project to propose semi-automatic tools for generating descriptions in both of the ways described above, according to the choice of the user. Before facing this goal, it is however necessary to propose the form of the description of the differences, which is still under research.

2.2.3 Converting externalized state of a component

The ability to specify conversions of externalized state of a component should be main benefit of utilizing a component implementation description. Dynamic component updating relies on the ability to perform transition of state between different versions of a component, here the most perspective approach is to transform the state when externalized in a persistent store.

The rules for performing such a transformation can, at least partially, be derived from the descriptions of old and new versions and their differences. The goal of our research is to propose such a framework for describing changes between implementations, which would minimize the user interaction required to perform the transformation.

This includes generating an initial description of changes by comparing the two descriptions of implementation. There are some other tasks which could be automated, e.g. generating a description of the implementation by examining the object's implementation written in an implementation language (*java*), but this functionality should rather be implemented by an environment built atop of the update description language.

2.3 Goal of the paper

The goal of this paper is to present the current results of the development of an *Update description language (UDL)*, suitable for describing changes made to either to the interface of a component, or to a component's implementation.

3 The CIDL language

To allow for describing changes between pairs of implementations, a *Component implementation description language (CIDL)* had to be developed first. Because it is not within the scope of this paper to provide a full description of the CIDL language, only an overview is provided and more focus is put on application of the language in providing a door to the externalization of components.

3.1 Introduction

The *CIDL (Component implementation description language)* is used to describe the implementation of a component. This description is intended to be independent on the chosen implementation language, however, an assumption is made that the implementation language is an object oriented one.

[2]The compatibility relation is still under development, although a partial definition shall be provided below

It is assumed that all state information is kept in objects. Although this assumption might sound too daring, it is not difficult to achieve. Although many languages allow global variables, it is possible to "cover" these variables in the CIDL description by an object having an attribute for each of the global variables.

Another, much more serious objection, is, that data is not the only part of the components state — it is formed also of states of all threads executing in the component. Nonetheless, even this obstacle can be dealt with. It has been assumed that externalization is performed only when the component is in a *well–defined state*, and one of the conditions of the well-definedness is that there are no pending requests made to this component. This solves the question of external threads making calls to the component, and we only have to deal with internal threads of a component. These surely can contain a significant part of the component state, however, they're to be managed by some object performing the ask of an *thread manager*. Then, the state of the internal threads can be considered a part of the state of the thread manager, and it is the responsibility of the thread manager to store the state of all the internal threads (and terminate them) when storing its own state.

3.2 Structure of the language

The language is strongly inspired by OMG's *Objects By Value (OBV)*. For each object type instantiated in the component, a corresponding declaration must occur in the CIDL description. This declaration shall list all attributes of the object, and can be used to generate code performing the externalization and restoring. In cases where it is necessary or appropriate, a custom method for performing these operations can be provided.

These declarations of object types are later used in *primitive architecture* definitions, which use a form similar to definitions of architectures of composed components. From these, declaration of object types are referred to using a **implementation** clause.

Example:

```
value TellerData; // forward declaration

value MyTeller supports TellerInterface {
  long Requests Served;
  TellerData privdata;
  transient SupervisorAccess su; // object reference
  DataStoreAccess ds;            // also obj. ref
};

primitive architecture MoonSoft : BankDemo :: TellerComp rev 1.1.1
     version "10-prim" {
    implementation BankDemo::MyTeller TImpl;

    bind Teller to TImpl;
    bind TImpl : dsReq to DS:ds;
    bind TImpl : suReq to SU:su;
};
```

3.3 Storing and restoring state of a component

The mechanism proposed for externalization of the state of components utilizes the *externalization by reachability* concept. One or more objects are chosen as an *externalization root*, and component is externalized by traversing the *object graph* rooted in these objects, and externalizing state of all objects encountered. State of objects is accessed either by using the general mechanism using information provided in a CIDL description, or by a user provided method for objects for which the default mechanism is not possible or suitable.

The component is externalized into a *persistent store*, in which it can be manipulated, transfered to another node, and from which it can be later restored.

4 UDL — Changes in the *component border elements*

Component border elements are CDL types, interfaces, and component frames. They are considered together, because change to any of them results into a changed (different) component frame, and therefore incompatible component architectures.

Although the original idea of SOFA/DCUP was, that only component architectures can change, the reality shows, that it is not the case. Even when doing the most careful analysis possible, the needs of the component's users change, and from time to time, it is necessary to extend and/or modify the interface of a service.

Even when only extending the interface, complexity of the interface specification (e.g., methods taking a parameter which type is a reference to the same interface) can easily result into non-compatible interfaces.

Also, modification of an interface results into incompatibility of types derived from this interface. Therefore, a mechanism for creating adaptors for connecting between different versions of a component border elements is essential.

Below, adapting mechanisms for all the three categories of component border elements shall be presented:

4.1 Interfaces

An UDL declaration describes the changes made when developing an interface (I_{new}) as a new version of an I_{old} interface. In principle, I_{new} is created by iteratively modifying I_{old}, however, compatibility can be declared even for type which were not derived one from another.

For the purposes of defining compatibility, we introduce the following formalism:

Assuming I to be an interface,

$super(I)$ refers to the set of its predecessors in the inheritance hierarchy,

$methods(I)$ refers to the set of methods of I, $methods(I)$ consists of methods declared in declaration of I and in declaration of all its predecessors,

$M \in methods(I), M = m(d_{p_1}p_1, d_{p_2}p_2, \ldots d_{p_n}p_n)$, is a method signature of a method m, where $n = nparams(M)$ is the number of parameters of m, $i \in \{1..n\}$, p_i is the type of the method parameter i and d_{p_i} is the parameter passing mode modifier ($d_{p_i} \in \{\mathbf{in}, \mathbf{inout}, \mathbf{out}\}$).

Note: For the purposes of keeping the definitions readable, we assume the return value of a method to be represented by an **out** parameter.

The UDL declaration defines the following hierarchy of modes of compatibility:

Compatibility by inheritance: It is possible to declare a new interface by inheriting from its previous version. The typing rules for inheritance ensure, that these types are also compatible by matching compatibility (and all compatibilities below). When new interface is declared this way, no UDL declaration is necessary, because their compatibility is obvious already from the CDL descriptions. However, this approach adds the old version into the inheritance chain of the new version, and in consecutive use, this technique does not scale well.

Definition:$compatInh(I_{old}, I_{new}) \stackrel{\mathrm{def}}{=} I_{old} \in super(I_{new})$

Example:

```
interface TellerInterface rev 1.5 extends TellerInterface rev 1.4 {
    add void QueryBalance(out float balance);
};
```

Matching compatibility: The set of method signatures of the new interface is a superset of the set of method signatures of the old interface.

Definition:
$compatMatch(I_{old}, I_{new}) \stackrel{\mathrm{def}}{=} methods(I_{old} :\subset methods(I_{new}) \ \wedge \ super(I_{old}) \subset super(I_{new})$

Example:

```
interface TellerInterface rev 1.5 compatibly modifies
  TellerInterface rev 1.4 {
     add void QueryBalance(out float balance);
};
```

Forwarding compatibility: All method calls performed on I_{old} can be *compatibly forwarded* to corresponding methods on the new interface $compatM(I_{old}.M, I_{new}.M)$.

Definition:

$$compatForward(I_{old}, I_{new}) \stackrel{\text{def}}{=} (\forall\ M \in methods(I_{old}) : \exists\ N \in methods(I_{new}) : canForward(M, N)) \ \wedge\ \ \forall\ S \in super(I_{old}) : compatType(S, I_{new})$$

$$canForward(M, N) \stackrel{\text{def}}{=}: M = m(d_{p_1} p_1, d_{p_2} p_2, \ldots d_{p_n} p_n), N = m(d_{q_1} q_1, d_{q_2} q_2, \ldots d_{q_k} q_k) \Rightarrow nargs(M) = nargs(N)\ \wedge\ \ \forall i \in \{1..n\} : (d_{p_i} \leq d_{q_i}) \wedge (ident(p_i, q_i) \vee (d_{q_i} = \mathbf{in}\ \wedge\ typecompat(p_i, q_i)) \vee (d_{p_i} = \mathbf{out}\ \wedge\ typecompat(q_i, p_i)) \vee (d_{p_i} = \mathbf{in}\ \wedge\ d_{q_i} = \mathbf{out}))$$

The relation $typecompat(p, q)$ is type compatibility relation, considered for all types defined in CDL.

$$typecompat \stackrel{\text{def}}{=} compatInh \cap compatMatch \cap compatForward \cap compatAdapt$$

Example:

```
interface Account rev 1.1 extends Account rev 1.0 {
    // some new methods
};

interface TellerInterface rev 1.5 compatibly modifies
  TellerInterface rev 1.4 {
    delete CreateAccount(out Account rev 1.0 account);
    add    CreateAccount(out Account rev 1.1 account);
};
```

Adapted compatibility: Although the interfaces were not compatible (not according to any of the definitions above), a method call mapping has been provided, and therefore these interfaces have been added to the relation $compatAdapt(I_{old}, I_{new})$.

Definition:$compatAdapt(I_{old}, I_{new}) \stackrel{\text{def}}{=} \exists\ A \in Adaptors : A = adaptor(I_{old}, I_{new})$

Example:

```
interface TellerInterface rev 1.5 modifies TellerInterface rev 1.4
               adapt_via ClassRef {
     add void QueryBalance(out float balance);
};
```

4.2 Types

For types, constructs syntactically similar to those used for interfaces can be given, extending the relation *typecompat*. Once are types (e.g., t_1, t_2) declared to be compatible, methods of otherwise incompatible interfaces (e.g., I_1, I_2), differing only in types of method parameters, which have now been declared to be compatible, become compatible. E.g., $m(\mathbf{out}\ \ t_1) \in methods(I_1) \wedge m(\mathbf{out}\ \ t_2) \in methods(I_2)$

The exact structure of the syntactic constructs vary with the kind of the type (which can be a **union**, a **struct**, a **enum** or a **typedef** referring to one of the previous ones, an interface, or possibly an array of these.

For interfaces, the main concern of compatibility was the ability to pass the method calls. Types are used as parameters in the method calls, and the main concern is in the ability to convert the data contained in one version of the type into the new one.

Example:

```
struct TellerParams rev 1.2 compatibly modifies TellerParams rev 1.1 {
   add string Title = "";
   add date EmployedSince = 0;
};

interface TellerInterface rev 1.6 compatibly modifies
  TellerInterface rev 1.5 {
    delete getParams(out TellerParams rev 1.1 account);
    add    getParams(out TellerParams rev 1.2 account);
};
```

4.3 Frames

For frames, no compatibility is declared. Compatibility of frames is not considered. Because each component implementation (architecture) is bound with exactly one frame, it does not make sense to judge whether an architecture is compatible with some other frame.

However, when an interface is modified, and a new version of a component frame is declared, it may become necessary to replace the frame in its parent component to allow for using an architecture implementing the new frame.

This shall be accomplished by modifying the architecture of the parent component, and reconnecting the bindings in the parent component from the instance of the old frame to an instance of the new frame, possibly via some adaptors.

UDL provides constructs for creating the new versions of the frame, and assists in building architectures using the new frames, inserting adaptors where necessary.

```
  // modifying a frame
  frame TellerComp rev 2.1.1 modifies TellerComp rev 1.1.1 {
     delete Teller;
     add provides: TellerInterface rev 2.1.1 Teller;
     // refresh a provides interface to a newer version
  };

  // modifying a architecture
  architecture MoonSoft : BankDemo::Bank rev 1.1.1 :
   version 2 modifies version 1 {
    delete Tellers;
    add inst TellerComp rev 2.1.1 Tellers;
    rebind providedTeller to Tellers : Teller;
    // uses an implicit adaptor
  };
```

In the above example, a change in the architecture was performed. However, this change description only aids to create a new architecture, which would allow for using subcomponents implemented with new `TellerInterface` in applications built with the old interface. This change does not say anything about transferring the component's state into the new version. These issues shall be focused on in the next section.

5 UDL — conversion of externalized state of a component

In the previous section, assembling application from components using various versions of component border elements was the focus. In this section, providing facilities for converting state between various versions of a component's implementation is the key topic.

There are various ways in which a component's implementation can change. A component's implementation can be either primitive or composed. It is essential to provide an ability to update from a primitive implementation to a composed and vice-versa, although such an update can potentially be very complex. Also note that, even if the type of implementation remains the same, the quantity and kind of changes can vary. For a primitive component, the changes can range from a change in method implementation through a minor data–format correction to a completely different implementation. For a composed component, the changes can range from replacement of a subcomponent with a subcomponent implementing an only slightly modified frame, possibly requiring a connector, up to a complete reconfiguration including splitting functionality (and state) of a single subcomponent into several subcomponents.

Although the complexity of changes can vary at a large scale, the facilities provided should allow *any* kind of update. For simple updates, the amount of information required from the user will be low, with growing complexity, this amount will increase.

An *update syllabus* describes steps necessary to transform an entity (which can be a component architecture, an implementation object, or a general data attribute) from an older version into a newer one. In such a syllabus, sub-entities of the respective entity can be updated by referring to some other syllabus. This way, a whole graph of state information can be easily processed.

In a syllabus, the following operations are allowed:

1. transforming an entity using an already declared update syllabus
2. deleting an entity
3. creating a new entity, with *initializing expressions* provided
4. changing the value of an entity

In initializing expressions, both old and new entities can be referenced. The UDL language provides a simple grammar for these expressions, more complex calculations can be performed by user supplied *updating objects*, which methods can be called from within the expressions.

Example:

```
update object BankDemo::MyTeller rev 1
    to BankDemo::MyTeller rev 2 {
  add string name = "";
  add float salary = blank;
};

update architecture ( MoonSoft : BankDemo::TellerComp rev 1.1.1 : "10-prim" )
        to ( MoonSoft : "1" ) {

   add inst  MoonSoft : BankDemo::TellerComp rev 2.1.0 "1"  NestedTeller;
   // although in the architecture section, the subcomponent instances are entered
   // without exact architecture specification, here is this information essential
   update TImpl from rev 1 to rev 2;
   add NestedTeller.TImpl = TImpl;
   delete TImpl;
};
```

5.1 Updates automation

Although there is no general algorithm for generating update syllabi, some (or possibly most) of the work can be automated — by examining a new implementation of a component, the CIDL description might be generated, and then by comparing the new description with the old one, a basic skeleton of the update syllabus might be created.

However, the appropriate place for this functionality is not the UDL compiler. Rather, a framework should be built atop of UDL, which would interact with the user, and serve as an *integrated development environment*. Further research is planned on this issue.

6 Related work

Maintaining application consistency during and after updates is a hot research topic, and some research groups focus even on describing updates being performed at application runtime. However, none of them address the concepts of transferring state between old and new version of a component during a runtime update. Unfortunately, even the issue of updating application at runtime is rarely addressed.

Dynamic Wright [ADG99] (see [AG94a] or [AG94b] for more information on original Wright project) focuses on dynamic application reconfiguration, however, this project is limited to a set of configurations known at compile time.

The PJama project [Dmi98] has solved the question of updating the persistent state of an object to conform to the new implementation of its class. Although this project does not consider runtime updating, it has been inspirative to us when developing UDL.

The CORBA Component Model [CCM99], a standard now being developed by the OMG, is dealing with component persistence. Our CIDL language has been partially inspired by the CORBA Component Model's component implementation description language. Because CORBA and SOFA use similar techniques for storing state of components, it might be possible to develop an updating mechanism for CORBA based on the results of our research.

7 Current status

The requirements put on the UDL language in the SOFA/DCUP project have been identified, and most of them have been at least partially dealt with. A basic chart of the UDL language has been proposed, on both the fields of the syntax and semantics of the language.

However, exact specification of the language has not been developed yet. Also, no implementation of the externalization mechanism necessary for component updating has been created so far. Both of these issues will be subject of further research.

Although the research is performed in an only locally known environment (SOFA/DCUP), it can contribute to development of any other component oriented framework and potentially, it could even aid in creating a software updating mechanism for the CORBA environment.

8 Future work

There are many areas to focus on. The most important ones are finalizing the language specification, and creating a pilot implementation of a CIDL and UDL compiler. Also, the externalization mechanism will have to be implemented.

After specifying the UDL language, effort will be put to propose and develop an integrated development environment, which would allow component developers to use the UDL framework comfortably and without entering redundant information.

References

[ADG99] R. Allen, R. Douence, D. Garlan: *Specifying and Analyzing Dynamic Software Architectures*, `http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/wright-fase98.html`

[AG94a] R. Allen and G. Garlan, *Formal Connectors*, Technical Report, CMU-CS-94-115, Carnegie Mellon University, March 1994.

[AG94b] R. Allen and G. Garlan, *Formalizing Architectural Connection*, In Proceedings of the Sixteenth International Conference on Software Engineering, pages 71-80, Sorrento, Italy, May 1994.

[CCM99] OMG: *The CORBA Component Model*, orbos/99-02-05

[CDL98] Vladimír Mencl: *Component Definition Language,* Master Thesis, April 1998, Charles University

[DO] Darwin Overview, `http://www-dse.doc.ic.ac.uk/research/darwin/darwin.html`

[Dmi98] Misha Dmitriev: *The First Experience of Class Evolution Support in PJama*, The Third International Workshop on Persistence and Java(PJW3), Tiburon, September 1998

[IDLspec] OMG: *OMG IDL Syntax and Semantics*, `ftp://www.omg.org/pub/docs/formal/98-02-08.ps`

[LKA95] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann: *Specification and Analysis of System Architecture Using Rapide*, IEEE Transactions on Software Engineering, pages 336-355, April 1995.

[LV95] D. C. Luckham and J. Vera *An Event-Based Architecture Definition Language*, IEEE Transactions on Software Engineering, pages 717-734, September 1995.

[Med96] N. Medvidovic. *ADLs and Dynamic Architecture Changes.* In A. L. Wolf, ed., Proceedings of the Second International Software Architecture Workshop (ISAW-2), pages 24-27, San Francisco, CA, October 1996.

[Medv97] Neno Medvidovic: *A Classification and Comparison Framework for Software Architecture Description Languages*, Technical Report UCI-ICS-TR-97-02, 1997

[OMG96] OMG 96-03-04, *The Common Object Request Broker: Architecture and Specification*, Revision 2.0, July 1995

[PBJ97] F. Plasil, D. Balek, R. Janecek: *DCUP: Dynamic Component Updating in Java/CORBA Environment*, Tech. Report No. 97/10, Dep. of SW Engineering, Charles University, Prague.

[PBJ98a] F. Plasil, D. Balek, R. Janecek: *SOFA/DCUP:Architecture for Component Trading and Dynamic Updating*, In Proceeding of ICCDS'98, May 4-6, 1998, Annapolis, Maryland, USA

[PBV99a] F. Plasil, M. Besta, S. Visnovsky: *Bounding Component Behavior via Protocols*, Accepted for presentation at TOOLS99

[Rap96] Rapide Design Team: *Rapide 1.0 Language Reference Manual*, Program Analysis and Verification Group, Computer Systems Lab, Stanford University, January 1996.