Autonomous Points in Component Composition[*]

[Extended Abstract]

Vladimír Mencl
Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic
mencl@nenya.ms.mff.cuni.cz

ABSTRACT

Current technologies provide only a primitive support for managing configuration of applications composed of software components; advanced facilities are needed, namely for specifying the points in a component hierarchy suitable for initiating an update or acquisition from an independent vendor. *Autonomous points* proposed in this paper reflect these requirements.

Keywords
software components, updating, architecture description languages (ADL), software configuration management

1. INTRODUCTION AND MOTIVATION

Let us consider an application composed of software components and focus on the stages of the application's life-cycle after its deployment. The application is likely to be updated and its configuration to be modified; however, current technologies provide only a primitive support in this respect, addressing only trivial cases (replacing the whole application, replacing only a single library, . . .).

Advanced updating issues are not addressed; this is because of not all components in the application/component hierarchy are suitable for initiating an update. Such a suitability is determined by the depth of specification of the respective component's interface. Components used only by components from the same vendor might not have a publicly available specification and therefore updates of these components should not be initiated by an unauthorized body.

A mechanism for selecting component interfaces suitable for initiating an update should exist. Moreover, to recover from an unsatisfactory update, a history of updates and previous configurations should be kept. The update mechanism should preserve properties analogous with the ACID transaction properties; in addition, backtracking to consistent points in the update history should be supported.

1.1 Background

In our work, we assume a *hierarchical component model*, i.e., a component model allowing composition of compound components. Such a component model should specify a black-box view of a component (the component's interface, consisting of services provided and required and potentially other specifications, such as behavior description). Also a grey-box view, describing the structure of a compound component at the first level of nesting, might be considered.

The propositions made in this paper are not bound to a specific component model; they only assume the model characteristics outlined above.

2. COMPONENT LIFE-CYCLE VERSUS SPECIFICATION

Several stages of component life-cycle are generally recognized [5, 4]; they usually include *development*, *assembly*, *deployment* and *run-time*, though other stages are sometimes also considered (e.g., *distribution* [2]). We will focus mostly on the assembly stage, as the key activity of the assembly stage is the selection of concrete components to be used as subcomponents for components so-far described only as a grey-box.

Similarly to EJB [5], the architecture roles involved at each stage of the component life-cycle are to be distinguished; also a specific configuration information related to each of these roles is to be provided. The key roles to be mentioned are the *component assembler* and *component deployer*. Though the selection of concrete components for subcomponents in elaborating a grey-box description is usually done at the assembly stage, different modes of component composition have to be introduced when independent service providers or third-party components are considered. For certain composition modes, the component selection may be postponed to later stages, most typically to the deployment stage, but also, e.g., till the runtime stage.

3. REQUIRED DEPTH OF COMPONENT SPECIFICATION

In the component based software development research, it is generally agreed that a component's interface is defined by the set of services provided and required; these services are usually declared as a set of method signatures, usually in an *Interface Definition Language (IDL)* (e.g., CORBA IDL). Such specifications describe only the syntactic nature of components, however, a general need for semantic descriptions of components has also been widely recognized. Such semantic specifications usually describe the acceptable sequences of method invocations on the union of the sets

[*]This work was partially supported by the Grant Agency of the Czech Republic – project 201/99/0244.

of the provided and required interfaces; they usually take a form of regular-like expressions [3] or CSP description.

Specifications can be used either as a means of the application design *(design-specifications)*, or as a description of a ready-to-use software component for the purposes of using the component *(use-specifications)*. At the analysis/design time, the developers may benefit from dividing an application into a higher amount of fine-grained components, as it allows them to see the structure of the application in advance.

However, use-specifications based only on interface declarations and method invocations sequences are generally not deep enough to reliably decide on substitutability of a component, especially because they do not deal with finer details of the component's interactions, e.g., arguments passed to the method invocations. Such details are usually a part of the internal knowledge of the component vendor, which may be documented, but they are not explicitly made a part of the component's specifications.

Note that creating such detailed specification would be too costly on human resources (specifications automatically generated from code can merely provide the same information as the automata and/or regular expressions method invocation sequence descriptions would provide); a specification deep enough to reliably decide substitutability is rather likely to resemble an API specification.

Moreover, the component vendor might not be interested neither in creating nor in releasing the specification at all, provided that other facilities involved in the software development process assure compatibility of the vendor's components assembled together in one contiguous block.

For using design architecture specifications, the developers should not pay the penalty of having to specify the use-specifications at the highest available level, guaranteeing algorithmic decision on substitutability. Therefore, an ideal component model should support multiple levels of depth of component specifications. The level would vary with the intention of the developers to provide the component deployer with the option to replace a component.

4. AUTONOMOUS POINTS

The claims of the previous section are to be reflected in the component assembly infrastructure. The task of the component assembly infrastructure is to manage selections of concrete subcomponents, in typical models performed by the *component assembler* architecture role. These selections may be altered by the component deployer, but there's usually no means to differentiate between the modes of component composition. By analyzing possible scenarios of component updates, we found it useful to distinguish two special cases:

1. Components may be provided as standalone services with a well defined interface (including also, e.g., QoS parameters). In such a case, the service provider may wish to hide changes made to components realizing the service. As far as the agreed properties of the service are held, the provider should not be obliged to reveal the changes made.

2. Component developers may wish to allow custom replacements of designated subcomponents, e.g., to provide a point for customization of the application, or integrating a third-party component.

Both these requirements can be satisfied by postponing the selection from the assembly to a later stage, either deployment or the runtime stage. We use the term *autonomous points* for those components in the application/component hierarchy, for which the selection of a concrete subcomponent is postponed to a later stage (and the responsibility for that action is transferred to a different role, either the component deployer or a runtime manager). In the full poster text, we intend to provide a technical description of the solution in SOFA.

5. EVALUATION & CONCLUSION

We pointed out the need to support multiple levels of depth of component specifications. Only designated components in the application/component hierarchy should be allowed to postpone the selection of concrete component to a later stage in the component life-cycle, we denote them as *autonomous points*. A proof-of-the-concept realization of these principles exists for the SOFA component model [1].

6. REFERENCES

[1] Plášil, F., Bálek, D., Janeček, R.: *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*, Proceedings of ICCDS'98, May 4-6, 1998, Annapolis, Maryland, USA, IEEE CS Press 1998

[2] Bálek, D., Plášil, F.: *Software Connectors and Their Role in Component Deployment*, Proceedings of DAIS'01, Krakow 2001, Kluwer 2001 (in print), 18 p.

[3] Plášil, F., Višňovský, S., Bešta, M.: *Bounding Component Behavior via Protocols*, In proceedings of TOOLS USA '99, pp. 387-398, Aug 1999.

[4] Object Management Group: *CORBA Component Model Specification*, orbos/99-07-01,

[5] Matena, V., Hapner, M.: *Enterprise JavaBeans 1.1 Specification*, Sun Microsystems Inc., August 10, 1999

[6] Medvidovic, N., Rosenblum, D. S., Taylor, R. N. : *A Language and Environment for Architecture-Based Software Development and Evolution*, ICSE-21, Los Angeles, May 1999

[7] Levans, G.T., Sitamaran, M.(eds): *Foundations of Component-Based Systems*, Cambridge University Press, 2000, ISBN 0-521-77164-1