Towards a Service-Based Architecture Description Language

Michael William Rennie[1] and Vojislav B. Mišić[1]

technical report TR 04/08
Department of Computer Science, University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
August 2004

[1]University of Manitoba, Winnipeg, Manitoba, Canada

Towards a Service-Based Architecture Description Language

Michael William Rennie and Vojislav B. Mišić

Abstract: The service-based computing paradigm is rapidly gaining ground as a viable choice for the design of modern software applications. In this paper, we review the basic operation of software components and the necessary infrastructure in the service-based paradigm, and identify the core set of requirements that those components and infrastructure have to satisfy. We argue that these requirements necessitate the use of a suitable architecture description language (ADL). Upon examining a few existing ADLs from the viewpoint of those requirements, we identify the salient features of a service-based ADL. In particular, we analyze the role of the core set of ADL elements such as supporting styles, validation facilities, and the representation aspect of architecture design.

Keywords: service-based computing, software architecture, architecture description language (ADL)

Contents

1 Introduction

Many approaches have been proposed to combat the ever-increasing complexity of modern software systems. Among the more promising approaches is the so-called service-based computing paradigm.

In traditional approach to software design and packaging, all the components of a software system are packaged together, even though most of them are used rarely, if at all. The users are forced to deal—buy, install, and keep on their disks—applications of ever-increasing size, even though they regularly use only a small portion of the available functionality, while many functions are used rarely, or ever at all.

The service-based paradigm is a more flexible approach in which software components are loaded on demand, i.e., only when their services are actually requested by the application, and removed when these services are not needed any more. Such components will be referred to as service components, in order to distinguish them from the more traditional view of components as independent entities in their own right [16].

The service-based paradigm did get an initial impetus from the development of Web Services, which may be considered as a family of service-based applications where (1) clients and service providers are permanently hosted on different computers, and (2) the interactions take place by exchanging messages written in an XML-compliant language over the Internet (i.e., using the TCP/IP family of protocols) [6]. (There is another important characteristic of Web Services—in their current form, at least—which will be discussed in more detail later.)

Despite their perceived importance for e-business and other applications, Web Services are but an instance (albeit an important one) of the service-based paradigm. In the most general case, there is no restriction on where the components acting as service providers and users may reside and where they may execute. Software systems in such diverse areas as office applications, autonomous systems, and intelligent agents can benefit from the service-based design. In this manner, many aspects of software system development, maintenance, and use could be simplified:

- As each component implements only a portion of the required functionality, it can be smaller and, therefore, easier to develop, test, and maintain.

- The memory requirements, and consequently the execution speed, of software applications can be reduced. Namely, an application need contain only the core functionality, while other, more exotic features, can be loaded and run only when needed. (On the other hand, the process of loading the component may take more time than if the component is hardwired in the application.)

- As the components that provide services are loaded on demand and removed afterwards, the process of updating the functionality of an application can be simplified. New versions of those components can be substituted as soon as theyre developed; whenever the user needs specific service or services, she will access the most up-to-date version.

- Important non-functional features such as security and quality of service can be dealt with in a consistent and dependable fashion.

- Finally, more flexible pricing schemes can be devised and implemented, in which the users pay only for the services (i.e., functionality) they actually use, rather than for the mammoth-sized applications with the functionality they dont need and never use.

However, advances in several areas are needed before the service-based paradigm can find widespread use for mission-critical applications in industry. One such advance is needed in the way software applications are designed: namely, the service-based paradigm should be adhered to in the design right from the start, rather than applied later as an afterthought. An important step towards that goal is the development of service-oriented architectural description languages (ADLs). The availability of suitable ADLs will not only enable the designers to specify the architecture of service-based applications in sufficient detail, but also to evaluate the design alternatives and validate them against the requirements, thus providing a firm foundation upon which service-based applications can be developed and implemented.

The paper is organized as follows. In Section 2, we review the characteristics of some existing ADLs that support dynamism, while in Section 3 we describe the operation of service-based software applications in greater detail, and outline the requirements that elements of such applications have to satisfy. Section 4 compares what we have with what we need, outlines the requirements for of a service-oriented ADL, and discusses its future implementation. Finally, Section 5 concludes the paper.

2 What we have: languages that support dynamism

An obvious starting point for this research is the existing body of work related to dynamic and evolutionary software architectures. The area of software architecture in general has attracted much research attention in the last decade [2, 15], and a number of ADLs have been proposed [3, 12]. However, the major part of this attention has been focused on problems related to static architectures, while the (much more interesting) problems of dynamism have yet to receive the attention it deserves. Despite some early attempts (e.g., [14]), only a handful of architecture description languages provide support for dynamism: most notably, C2 [11], Darwin [8], RAPIDE [7], Weaves [13], and Wright [15]. (Note that, strictly speaking, C2 is the architectural design paradigm, while the name of the language is C2SADL; for brevity, we will refer to the language simply as C2.) The service-based computing paradigm has been introduced recently, and no language that we know of deals specifically with its features. In the discussion that follows, we will briefly review the basic concepts of ADLs, and then proceed to analyze the support for dynamism in Darwin and C2 as representative ADLs that allow constrained and unconstrained dynamism, respectively.

2.1 Core Requirements

An ADL is a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation [12].

The features may be classified in different ways. From the user standpoint, we may distinguish between three not quite disjoint categories: end-system-oriented, language-oriented (i.e., those of ADL itself), and process-oriented features (i.e., those pertaining to the actual usage of the ADL to create and validate architectures) [9].

From the structural standpoint, an ADL must provide support for the following elements of an architecture [12]:

- Components, such as types, constraints, interfaces, evolution and semantics;
- Connectors, including interfaces, types, constraints, evolution and semantics;

- Architectural configurations fostering understandability, compositionality, heterogeneity, constraints, dynamism, scalability, refinement and traceability and evolution; and
- Tool support for code generation, dynamism, active specification, multiple views, analysis and refinement.

It should be noted that the basic items listed here are not independent of each other, which does not make the task of ADL design any easier, of course.

Support for dynamism in ADLs includes structural issues such as parameterized instantiation of architectural elements, component and connector replication, and conditional reconnection (among others), but also semantic issues such as dynamic configuration semantics and architecture recombination events [9].

Furthermore, dynamism may be constrained, i.e., known and specified at design time, or unconstrained, in which case the bindings may be determined at run-time without being specified first. Of course, any valid constraints on the components and connectors that are bound must hold at all times, regardless of whether the actual binding has been specified in advance or not.

Let us now review the two ADLs mentioned above, and the focus on their support for dynamism will serve as a good introduction to the issues related to service-based software design.

2.2 Darwin

Darwin [8] encompasses most of the elements that need to be considered in an ADL. It does fall somewhat short, however, when the requirements of the service-based paradigm are taken into account.

Darwin supports components and connectors, using subclassing to build more specific components from generic ones. Each component is described in terms of services it provides and services it requires. Darwin also allows composite components obtained from instances of simpler ones.

Individual services are specified using the π-calculus, while the configuration is specified by binding services requested by one component with services provided by another one. (As mentioned above, a binding can occur only if the type of the provided service matches the type of the required one.) Thus, the specification of the structure is decoupled from the specification of individual components and their services, as required by the principle of separation of concerns.

Darwin allows components to be instantiated, and their services bound, either statically or at run time. In the latter case, two options are available. In the so-called lazy instantiation, a component is not instantiated until another component wants to use its services. However, the types of participating components and the binding(s) between them must be specified beforehand. In this manner, the system can evolve at run time, but the manner in which it evolves is fixed at design time, and no cycles are allowed.

The other option, known as direct dynamic instantiation, allows arbitrary structures to evolve at run time. However, if the dynamically instantiated components want to interact, the available options are limited. In particular, if such components are to interact directly, they must exchange the relevant information (i.e., service references) through a third party, which is not part of the language per se.

In either case, bindings are permanent and cannot be undone. The goal of Darwin designers was to keep the notation declarative, and the introduction of unbind operator would violate that requirement. In other words, architectures specified in Darwin may grow (subject to certain restrictions), but they cannot shrink. Furthermore, the communication between system elements is subject to restrictions, and the help of an external agent may be required to achieve the full potential of an architecture specified in Darwin.

These restrictions make Darwin rather unsuitable for the design of service-based architectures, where both binding and unbinding on demand are required features, as will be seen in subsequent discussions.

Nevertheless, some of the features of Darwin make it a useful starting point for the specification of service-based architectures.

2.3 C2

C2 is one of the few ADLs hat support fully unconstrained dynamism [11]. In C2, both components and connectors are first-class entities, and message passing is used to link components to one another through connectors, subject to certain restrictions. Furthermore, connectors may perform optional filtering of messages.

C2 allows dynamic binding (referred to as welding in C2 parlance) of components to connectors, as well as unbinding (unwelding). All possibilities for reconfiguring or rewiring of an architecture are allowed: it is possible to unweld one component from another, reweld it to a different one, leave it unwelded and persistent within the system, or remove it from the system entirely.

An interesting feature of C2 is that dynamic reconfiguration is accomplished using message passing between components, i.e., in the same way as the ordinary communication between them.

The Architecture Construction Notation (ACN) and the associated infrastructure C2 provide an API interface to help marshall the welding/unwelding process between components. CS also supports the upgrading of existing components through subtyping, modifying the subtype, adding the subtype to the architecture, and then removing the original component from it.

Yet even with ACN and the associated API in place, C2 is not without its share of troubles. In particular, a number of issues are not resolved in their entirety, i.e., how to unweld components that are currently in use (as their unwelding may cause loss of messages in the system) or those that have dependants (as it is not clear what to do with them), how to initialize the newly added and welded component, and how to ensure the consistency of the architecture throughout the structure changes (which may be quite extensive).

While the unconstrained dynamism of C2 certainly fits well the requirements of the service-based architectures, the need to have a separate external facility such as ACN in order to provide the functionality necessary to achieve dynamism points to an important aspect of dynamic architectures and corresponding ADLs: namely, that full support for dynamism cannot be achieved without proper tool and infrastructure support.

2.4 Tool and infrastructure support

Automated tool support is considered to be an essential part of any ADL [12]. Of course, different phases of the system lifecycle require the use of different tools:

Design tools are needed in the architecture design phase. They should provide editing and documenting facilities, but also design guidance and criticism, as well as correctness and consistency checking.

If our design is to be based on several existing ADLs, perhaps a tool that provides interoperability, such as Acme [5], could be used.

Code generation tools are needed to convert the architecture from a design to an executable form. An example of such a tool is ArchJava [1], which allows architectures to be built in the form of executable Java code, with the focus on maintaining communication integrity.

Table 1: Options for location of service components and the manner in which they are accessed.

component residence	manner of access	
	local	remote
local	traditional or service-based	service-based (server-side)
remote	service-based (client-side)	service-based (incl. Web Services)

Runtime support tools may be needed to execute the code thus generated. In some scenarios, runtime support may be built in the generated code; in others, a separate runtime infrastructure is needed, or the necessary infrastructure may already be available on the execution platform. Middleware such as CORBA, COM, or RMI, have significant potential for facilitating the runtime support for service-based systems, although many problems, esp. in terms of interoperability and portability, remain to be solved [10].

It should be noted that the most likely vehicle to implement meta-data is XML, which is rapidly becoming de facto standard for data exchange, and many tools to create, manipulate, and store XML-formatted data are available [4].

Communication between the components may also use XML, but it is not mandatory. Namely, XML is probably a viable alternative for remote access to services, much like the solution adopted for Web Services. In systems where service components are available (and bound) locally, there is less need to rely on XML, as its flexibility is somewhat offset by the performance penalty it incurs.

3 What we need: requirements for service-based applications

Service-based applications operate in a rather different manner than their counterparts developed according to the traditional paradigm, as mentioned above. It should come as no surprise, then, that their requirements must differ as well.

3.1 Component location and access

Implementation-wise, there must be a common communication mechanism through which the interactions may take place. In simpler usage scenarios, service providers can reside on local fixed storage from which they may be loaded when necessary, not unlike Windows Dynamic Linked Libraries or their equivalents under other operating systems. The taxonomy of these options is given in Table 1.

In more complex scenarios, services can be accessed through a private or public network, such as the Internet. In this case, the service component can run at a remote site, similar to the approach adopted by Web Services, and its services can be accessed remotely; alternatively, the service component can be downloaded to the local site and installed from it. In the former case, the interaction will in fact consist of a series of messages exchanged between the client and the service provider. In the latter case, once the external service component is downloaded, it becomes indistinguishable from a local one. However, when the interaction is over, the component will be removed from memory, and possibly even deleted from local non-volatile storage.

Scenarios like those just described necessitate the presence of an infrastructure capable of managing the interactions described either as part of the original client application, or independently of it. Such an infrastructure may be embedded in the operating system and thus made available to all the applications, or it may be made to run as part of the actual application. (The choice, of course, depends on the facilities offered by the operating system.) By analogy with existing operating systems and with Web Services, we will refer to this infrastructure as the registry, and its operation will be described in greater detail in the discussion that follows.

There may be one centralized registry to support a given application, or perhaps a number of registries that will reside on different hosts and cooperate to provide the required behavior. In this context, to support means to keep track of the service components available and to manage their use as required by the application.

Finally, the components that act as service providers must be aware of their own capabilities, or at least contain sufficiently rich meta-data so as to enable the selection of the most suitable service for a particular client query. But before we discuss the required meta-data in more detail, let us dwell for a moment on the process of service search and selection.

3.2 Service search and selection

Let us now discuss the process of searching for services and selecting the most suitable one. As mentioned in the Introduction, the mechanism of locating and selecting the most appropriate service to perform the required task is the third cornerstone of the Web Services paradigm (the other two are the use of Internet-cum-XML for messaging, and remote activation and access to the services involved, again through XML)

In the Web Services approach, services to be accessed are located via their signatures and simple descriptions written in WSDL [6]. This approach is essentially the same one which has been in use in various programming languages for almost fifty years. While simple and efficient, it suffers from a major drawback: namely, it requires the designer to know in advance the signature of the service to be invoked. By extension, it means that the client must possess detailed knowledge of the component that will provide the service.

This, however, does not fit well with the dynamic and ever-changing nature of service-based applications. Service components are not known beforehand and they should be accessed on the basis of their advertised capabilities for providing particular services, rather than according to the name and signature. To draw an example from a rather different area: when we go to the bank to deposit or withdraw money, we dont really care who the actual teller is, as long as she is able to fulfill our request.

To take this idea one step further, we propose a dynamic architecture in which the client component will be able to query the registry in order to find the service component that optimally satisfies its needs. Once such a component is found, the client can ask the registry to bring it to the local host and bind it to the client. Alternatively, the client may opt for remote access due to memory, timing, or security constraints, or the constraints dictated by the service component itself.

In terms of loading the required service components, traditional applications utilize static and (sometimes) dynamic loading, depending on the perceived need and usage frequency for a particular service component.

Service-based systems tend to rely on dynamic, on-demand loading of service components, which may or may not be available locally at the time they are needed. In both cases, service components locally bound and subsequent interactions are local.

We note that Web Services belong to the other end of the spectrum. First, they rely on remote access. Second, the UDDI service which acts as the Web Services registry provides just the matchmaking between clients and service providers, and does not have anything to do with the actual interaction between the two.

A simple taxonomy of options for to service selection and loading is given in Table 2.

Table 2: Options for selection and loading of service components.

loading	service component selection	
	by name	by service feature(s)
permanent	static	N/A
dynamic	dynamic (e.g., DLL)	service-based paradigm
none	WSDL (Web Services)	extensions to WSDL

3.3 Some features of the registry and service components

The implications of these requirements for the architectures designed to support them, as well as for the mechanisms (such as ADLs) to specify such architectures, are subtle but far-reaching.

The registry needs to be much more active than in the case of Web Services. First, it has to provide flexible query and selection facilities far beyond the rather rudimentary capabilities provided by the UDDI.

Second, the processes related to the actual downloading of the service component from its remote host, loading it to memory, and binding it to the client, have to be performed by the registry. Once the services of a particular component are not needed any more and it is unbound from the client (either by the client or by the registry at the request of the client), the registry has to remove the component from memory; alternatively, it can decide to keep it in some sort of service component cache, just in case its services are again needed.

Third, the registry has to monitor the execution of the service components so that it can react to errors, access rights violations, and other possible risks.

Finally, the registry has to keep track of the components under its supervision so that it can perform efficient and accurate searches when queried by its clients. In case of distributed systems, the registries can cooperate to execute users queries.

The service components also need to be able to do much more than just execute an algorithm and/or perform some input/output operations.

They have to register with their local registry and to provide a significant amount of information about themselves. The minimum information required consists of the names and signatures of the services provided by the component, as in traditional, statically bound applications (or in Web Services, for that matter).

However, in order to allow selection through more complex queries, the component may also contain the information about various features related to its use. The features that might be of interest include functional constraints (i.e., inputs and required outputs as in design contract), non-functional constraints (memory and timing requirements), security- and access-related features (e.g., may be downloadable), price-related statements (e.g., pay-per-use or subscription-based payment), and other relevant issues, as deemed necessary for the selection and/or use of that particular service component.

Since some of the service components may need to be downloaded from a non-local host, the components could also indicate how long theyre willing to wait for the service component to become available. If timing is not important, the decision may be left to the registry itself.

All of this information taken together indicates that the design of service-based systems may, at some future time, be capable of operation under defined quality of service limits.

Some of this information may be defined at design time and subsequently embedded within the component itself. The remaining part may be recorded and subsequently maintained by the registry itself, based on information obtained by observing the component in execution or by any other means.

The components, both those initially loaded as part of the application and those dynamically bound at a later time, need to distinguish between mandatory services and optional ones. The question is simply whether the failure of the load/bind request for a particular service will cause the abortion of the execution of the client, or the client will be allowed to proceed. For example, failure to print from an application should be reported but need not cause the application to be terminated; but failure to load a security module should lead to termination of an e-commerce application.

The service components should also be aware of any other service they might need in the course of their execution. These needs should somehow be reported to the registry; again, mandatory services should be distinguished from the optional ones. In this manner, the registry will be able to perform pre-loading of mandatory services, or report failure if some of these services cannot be loaded.

Service components which are no longer needed by any of the running applications will be unbound. The registry can then decide whether to remove them or keep them in some sort of cache for speedier access if they are again needed at a later time. (The idea of garbage component collection does spring to mind.) The decision might also be affected by other concerns such as pricing agreements or security considerations.

It is obvious that the specification of both the registry and the service components cannot be approached in the same manner as traditional, statically-bound software components. Let us now try to compare what we have and what we need, and try to outline the manner in which the existing ADLs can be modified to fit the needs of service-based application design.

4 The best of both worlds . . . ?

We can now compare what we have and what we need, and then use the conclusions to envisage an ADL that would be better suited for architectural design of service-based systems.

As for Darwin, the structure of the architecture is described in a rather simplistic but effective notation. The simple declarations in Darwin, or at least in Darwin-like style, and the separation of the descriptions of services and components from those of the architecture proper, are highly desirable features.

On the other hand, the instantiation and binding mechanisms provided in Darwin leave a lot to be desired; in particular, the insistence on its declarative nature must be abandoned, as it precludes the implementation of features (such as unbinding) which are sine qua non for service-based architectural design.

Some of the facilities offered in C2 are much better suited for service-based computing, despite its initial orientation towards graphical user interfaces [11]. In particular, the provisions for dynamism and architecture evolution are highly developed, even though some of the common scenarios are not entirely resolved within the C2 architecture or the associated ADL.

Regarding service component selection and loading, the approach used for Web Services should be adopted. However, it must be enhanced with rich meta-data and the alternate possibilities for selection, as explained above, if the full potential of the service-based paradigm is to be realized.

It goes almost without saying that the meta-data should be formatted using an XML-based notation. This will enable the use of sophisticated support tools which are currently available for XML. This will also allow the use of emerging query languages and facilities based on XML, such as XQuery, XSQL, and XSLT, and querying the registry is a crucial component of our proposed approach.

Regarding component loading and unloading, it should be done at runtime (in a hot-swappable manner), without having to stop the client application or the entire system. This facility is already present in commercial systems, such as DLLs in Windows and package addition/removal in Linux, where recent versions allow even the OS kernel to be updated at runtime.

Components without dependents can be unbound and removed without hassle. When the component to be removed has dependents, however, the registry should analyze those dependencies, not unlike the manner in which the mark-and-sweep garbage collection works, and decide on a case-by-case basis which of the components can be removed and which cannot. As mentioned above, the decision may also be affected by performance, pricing, security, or other considerations.

Finally, in both Darwin and C2, the implementation and deployment of architectures do require a rather sophisticated infrastructure. (This constraint is implicit in Darwin, but quite explicit in C2, through the introduction of ACN.)

5 … But much remains to be done

The report presents the work in progress, with the goal of designing an architecture description language and the associated infrastructure for service-based systems that would build upon the features of existing ADLs and experiences in using them.

We also plan to implement an experimental environment in which distributed service-based systems could be developed from the architectural design phase to the actual deployment, and thus provide valuable information on the design and use of such systems. Petri net-based formalism will be used to validate the operation of the environment and provide a convenient foundation for future work. We are also in the process of designing the actual ADL and the supporting infrastructure.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th international conference on Software engineering*, pages 187–197, May 2002.

[2] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. The SEI Series in Software Engineering. Addison-Wesley, Reading, MA, 2nd edition, 2002.

[3] P. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, Paderborn, Germany, Mar. 1996.

[4] E. Dashofy, A. van der Hoek, and R. Taylor. An infrastructure for the rapid development of XML-based architecture description languages. In *Proceedings of the 24th international conference on Software engineering*, pages 266–276, Orlando, FL, May 2002.

[5] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, pages 7–21, Toronto, ON, Nov. 1997.

[6] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services architecture. *IBM Systems Journal*, 41(2):170–177, 2002.

[7] D. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison Wesley, Boston, MA, 2002.

[8] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 3–14, San Francisco, California, United States, Oct. 1996.

[9] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops*, pages 24–27, San Francisco, CA, 1996.

[10] N. Medvidovic. On the role of middleware in architecture-based software development. In *Proc. 14th international conference on Software Engineering and Knowledge Engineering*, pages 299–306, Ischia, Italy, July 2002.

[11] N. Medvidovic, D. Rosenblum, and T. Richard N. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st international conference on Software engineering*, pages 44–53, Los Angeles, CA, May 1999.

[12] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan. 2000.

[13] P. Oreizy, M. Gorlick, R. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and Their Applications*, 14(3):54–62, May/June 1999.

[14] P. Oreizy, N. Medvidovic, and R. Taylor. Architecture-based runtime software evolution. In *Proc 20th International Conference on Software Engineering (ICSE98)*, pages 177–186, Kyoto, Japan, Apr. 1998.

[15] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Englewood Cliffs, NJ, 1996.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Harlow, UK, second edition, 2003.