Software Configuration Management Related to Management of Distributed Systems and Services and Advanced Service Creation

Vladimir Tosic[1], David Mennie[2], Bernard Pagurek[1]
[1] Network Management and Artificial Intelligence Lab
Department of Systems and Computer Engineering, Carleton University, Ottawa, Ontario, Canada
{vladimir, bernie}@sce.carleton.ca
[2] The Bulldog Group Inc., Toronto, Ontario, Canada
dmennie@bulldog.com

ABSTRACT
In this position paper, we briefly summarize three research projects, conducted at the Network Management and Artificial Intelligence Lab at Carleton University, related to software configuration management (SCM). These projects are dynamic service composition from service components, dynamic adaptation of service components with multiple classes of service, and dynamic evolution of network management software. We then summarize some of the challenges for future research in SCM. Our research group is particularly interested in two challenges: 1) managing dynamism and runtime change and 2) integration of SCM with other management areas and domains.

Keywords
Software configuration management, component configuration management, dynamic service composition, dynamic service adaptation, dynamic software evolution

1 INTRODUCTION

In the past, our research group focussed on challenges particular to the domain of network management including network configuration management. This included investigating the use of mobile agent technologies and artificial intelligence techniques to enhance how a network is controlled and managed. However, in recent years our focus has shifted more towards service management. Many of our recent research projects have uncovered issues that are related to software configuration management (SCM).

We view SCM as part of the broader work on the management of distributed systems, networks, applications, and services. The International Organization for Standardization (ISO) identifies configuration management as one of five network and systems management functional areas, which also include fault management, performance management, accounting management, and security management. On the other hand, the management of software (i.e., applications) is becoming more tightly connected with the management of computer systems (both stand-alone and distributed), peripheral devices, networking infrastructure, databases, and other resources. Integrated network and systems management – also known as enterprise management – strives to unify these management domains within the overall computing/communication system. A further unification is achieved with service management where services (software- and/or hardware-based) are managed from the business-centered (not technology-centered) point of view. This requires mapping technology-centered management activities and data into appropriate business issues and data directly showing how the value provided to end users and service providers' responsibilities stated in service level agreements (SLAs) are affected.

Some of our research efforts have concentrated on service components. A detailed discussion of how service components differ from software components is beyond the scope of this position paper. The central idea is a service component is a type of software component that provides some software-based services and optional hardware-based services like memory, printing, network bandwidth, etc. In short, the service component attempts to unify the concepts behind the software component and the distributed object. This means two or more service components, potentially distributed, can be composed together to form a single new service that can be deployed to an end user (human or software). Component service providers (CSPs) differ from application service providers (ASPs) since they do not provide monolithic applications and instead offer libraries of composable service components. Despite the differences, there are also many similarities between service components and software components. Consequently, we believe that there are important relationships that exist between the creation and management of service components and the work on component configuration management [4].

2 AN ARCHITECTURE TO SUPPORT DYNAMIC COMPOSITION OF SERVICE COMPONENTS

Dynamic (i.e., runtime) composition of service components focuses on adapting running applications and changing their existing functionality by adding new features or removing features, usually in an automated process. There are several key benefits to dynamic service composition. The most immediate is the fact that applications have greater flexibility since new services can be constructed to address a specific problem if they do not already exist. Another benefit is users do not need to be interrupted during upgrades or the addition of new functionality into a system. In other words, virtually an unlimited set of services can be created from a set of basic service components. Also, services can be assembled based on the demands of an application or a user. For example, if a user requires an Internet search engine that will filter out advertising from the results returned for a particular query, the service can be assembled at runtime and sent to the user. This service may not have been designed or even conceived ahead of time.

Composing service components at runtime is a challenging undertaking because of all the subtleties of the procedure involved, the many exceptions to the compositional rules that can occur, and the potential for error. The challenge lies in dealing with many unexpected issues in a relatively short period of time since all decisions must be made relatively quickly or dynamic composition becomes impractical. There is also a lack of support for dynamic techniques in programming languages and other development tools. In dynamic composition, as we have defined it, it is extremely difficult to predict beforehand the exact environmental conditions that will exist in a system at the time a composition is performed. We call this unanticipated dynamic composition [3], meaning that all potential compositions are not known and neither the service components nor the supporting composition infrastructure are aware if a particular composition will be successful until it is actually carried out. While steps are taken to decrease the chance of a failed composition, it cannot always be avoided. One of the measures we have taken to avoid complications is to bundle a service specification with each service component that describes the dependencies, constraints, or potential incompatibilities for the component. This specification also contains a list of the operations contained within the service component that can be reused in a composite component. These methods are referred to as composable methods. By looking at the specification for each component of interest before attempting to aggregate them in a composite service, failed attempts can be minimized or recovered from. The general rule followed is if a conflict is detected by the supporting infrastructure, the composition is aborted. Locating components at runtime for composition requires a component library or code repository that is integrated with the software infrastructure performing the composition.

The fundamental challenge in composing services at runtime is the design and implementation of an infrastructure that will support the process. Recently, we designed and implemented a general-purpose dynamic service composition architecture called the **I**nfrastructure for **C**omposability **A**t **R**untime of **I**nternet **S**ervices (ICARIS) [6]. The architecture provides all of the required functionality to form composite services from two or more service components that have been designed for composability.

There are three primary composition techniques that are supported in the architecture [5, 6]. First, creation of a composite service interface for several service components is achieved by extracting and combining signatures of their composable methods. Service components involved in the composition remain distinct, while communicating with clients through the common composite service interface. The composite service interface redirects all incoming calls to the appropriate service component for execution. Second, creation of a new stand-alone composite service is achieved by interconnecting service components using a pipe-and-filter architecture. In essence, the pipe-and-filter architecture chains the output of one service component to the input of the next. While this is a fairly primitive connection scheme, some complex constructions are also possible. One example is when the outputs of one component are looped back into its inputs. Other connection schemes, such as service components processing the same input in parallel, are not supported in the ICARIS architecture because most applications do not require such specialized configurations. Third, creation of a new stand-alone composite service with a single body of code is achieved by extracting and assembling the composable methods from software-based service components involved in the composition. The corresponding method signatures are also merged into a new composite service specification. This is the most challenging type of service composition and one motivation for its undertaking can be performance. In theory, a composite service with a single body of code may take longer to create than the other types of composite services, but it should also execute much faster.

A prototype application using a Jini, JavaBeans, and XML-based implementation of the ICARIS architecture was developed to illustrate a justifiable use of dynamic composition techniques. The prototype allows construction and deployment of security associations between a client and server in the network in order to enable security services to be introduced into applications that do not already have access to security. The specification of these security associations is carried out at runtime, so that the composite client and server security services can be constructed dynamically. The prototype is an accessible, robust infrastructure that is capable of establishing many types of security associations for any application. Additionally, it is both fast enough to assemble and deploy these associations at runtime and flexible enough to add or remove secure services to meet the applications it serves. A more detailed description of the motivation for and the characteristics of this prototype is given in [6].

3 THE CONCEPT OF SERVICE COMPONENTS WITH MULTIPLE SERVICE OFFERINGS

Another project in our research group investigates how to provide additional support in service components for dynamic adaptation and management of service compositions. The goal is to enable a composite service to adapt to many possible changes in its environment. We are particularly interested in the loose coupling of service components when the composed service components are not very dependent, they can be distributed, and one service component may be part of many different compositions and serve many different clients. This work is related to the work on dynamic service composition and to the work on adaptable software presented in [8, 9].

Service components should be designed so they can be used in a variety of different composite services. This allows a wider array of clients to take advantage of the services provided by the component. To achieve this, each variation of a service component can be represented by a class of service. A class of service is associated with a specific utilization of system resources and therefore has a different cost for the client. Differentiation of classes of services enables a client to receive the exact service and quality of service (QoS) it requires, within an acceptable price. It also allows the service component to balance its limited hardware and software resources efficiently. Examples of different classes of service include different usage privileges (e.g., access rights), different priorities, and different response times guaranteed to different classes of clients. Note that for software components the benefits of differentiating classes of service are not as strong as for service components.

The benefits of formal specification of constraints in component-based software engineering are widely recognized. For service components, it is necessary to formally specify functional constraints (preconditions, postconditions, and invariants), non-functional (QoS) constraints, and authorization policies. As an increased number of service components offering similar functionality become available, the QoS, price/performance ratio, and adaptability will become the major criteria that differentiate the service components from one another. Specifying functional constraints in a formal way and bundling the specification with the service component can improve the effectiveness of the dynamic service (and software) composition process. Formal specification of non-functional constraints is also important for application and service management because components that have different levels of QoS can be dynamically reconfigured or replaced with a less resource-intensive variant should system resources become limited. Finally, formal specification of authorization policies can be used for policy-driven management.

In order to provide adequate support for the previously discussed issues, we are investigating the processes of service creation and management using service components with a slightly different structure. In this case, a service component not only has multiple interfaces (units of service functionality) but it also has multiple service offerings representing different classes of service. Service offerings within an interface relate to the same functionality, but differ in authorization rights, QoS constraints, and cost. Occasionally, differences in functional constraints may exist between service offerings but this is generally not the case. By separating the concepts of an interface and a service offering, we provide additional support for dynamic flexibility and adaptability within a service component.

We specify service components (including all interfaces and service offerings) in a comprehensive XML-based service specification format that describes the functionality, the functional and non-functional (QoS) constraints, the authorization policies, and the cost of the different options. Such a comprehensive specification supports dynamic service composition. It enables finding at runtime appropriate service components and service offerings based on functionality requirements, constraints (functional, non-functional, authorization rights), and cost.

Dynamic adaptation mechanisms are also required to manipulate service offerings appropriately. These dynamic adaptation mechanisms include switching between service offerings, deactivation/reactivation of existing service offerings, and creation of new appropriate service offerings. Note that these mechanisms are under control of the service components and therefore their use can be restricted to specific clients. The mechanism for dynamic creation of new service offerings would be one example where access control is required. Dynamic creation of new service offerings will generally be used by a service component when its implementation is dynamically changed (e.g., in the case of dynamic versioning) or when the services that it uses are dynamically updated (e.g., offer better QoS).

Development of new service management algorithms based on manipulating usage of service offerings is suitable for several different situations. For example, if a service offering has to be dynamically deactivated while it is used by at least one client, maybe the same or similar functionality could be provided to the dependent client by a different service offering of the same service component. Another example is dynamic evolution of service components. We are currently developing such service management algorithms and appropriate architectural support for their implementation. Some of the issues that we try to address are: how to relate service offerings to better support automatic switching between them, how to integrate the support for deactivation/activation of service offerings into service components, how to support rules governing creation of new service offerings, etc.

4 DYNAMIC EVOLUTION OF NETWORK MANAGEMENT SOFTWARE

Our research group is also investigating the issue of the dynamic evolution of software without disrupting its operation. The evolution required can be corrective (fixing bugs or problems), perfective (improving performance or adding new functionality), and/or adaptive (adaptation to new operation environments) [8]. This is an important issue for high-availability and real-time systems. As noted in [8, 9], there are a number of different approaches taken by the research institutions that are trying to address this issue. The major efforts are based on the design of a software architecture, a new programming language, a data-flow architecture, a distributed system, a distributed object technology such as CORBA and COM, a compiler, an operating system, or a real-time system that will support software evolution. Some of the issues that pertain to dynamic software evolution are described in [2]. These include developing the appropriate infrastructure support, developing dynamically upgradable software modules, defining the module granularity, defining the scenarios where software upgrading is allowed or not allowed, obeying the limits on the allowed duration of the upgrading process, transferring the state between mo dule versions, and transactional issues.

We are particularly interested in the problem of dynamic software evolution as it pertains to network management software. A network management system is a relevant case study that can be used to capture general software evolution issues. Shutting down the entire network management system to perform an upgrade is not an appropriate solution for large mission critical networks. We have developed our own infrastructure and experimentally applied it to a modular SNMPv3 (Simple Network Management Protocol, version 3) system implemented in Java [2].

Our approach to dynamic software evolution, called software hot-swapping [1, 2], is based on the concept of swappable modules (S-modules) and corresponding non-swappable proxies (S-proxies). Only S-modules can be hot-swapped in our architecture. Each S-module has an S-proxy that is permanently associated with it and the S-proxy is automatically generated. The S-module and the S-proxy together constitute an S-component. Apart from S-components, there can be a number of other non-swappable modules in the application. An application supporting hot-swapping must contain a Swap Manager that controls all swapping transactions. Currently, all application modules that are expected to be hot-swapped have to be manually converted to S-modules prior to a swap. We have attempted to automate the process of converting application modules to S-modules and anticipate a solution in the near future.

When a new S-module version is transferred to the desired location by means of mobile code, its currently executing version has to be put into a swappable state before a hot-swap can take place. The S-proxy gets the state of the currently executing version, initializes the new S-module version with this state, and redirects all references from the current version to the new version. If during this process, the given swapping time limit is exceeded, the process is terminated and rolled-back. Otherwise, the new S-module version is started and the old S-module version is removed from the system. In order for the hot swap to succeed, appropriate support for hot-swapping has to be integrated into S-modules. This includes the ability to extract the state of a running S-module and to initialize an S-module with the state extracted from the previous version. The mapping rules between different versions have to be defined for every S-module and can be implemented as part of the S-module initialization methods.

Note that this proxy-based approach was chosen after a study of the advantages and disadvantages of several possible solutions. Further detail on this study can be found in [1]. While we found the proxy-based technique to be suitable and efficient enough for the dynamic evolution of SNMPv3 modules [2], its applicability to other software architectures is still under review. For some real-time systems that require even higher availability other approaches to dynamic software evolution may be more appropriate.

5 CONCLUSIONS AND CHALLENGES

Managing dynamism and runtime change is the most important challenge for SCM that we have focussed on in our research. The archetypal problem we are addressing is how to dynamically and autonomously (i.e., without explicit human intervention) reconfigure and, if necessary, upgrade software in order to minimize the effects of a fault (or a performance problem) on an end user. This problem is a very complex one with many open issues. We have broken the problem down into sub problems and our different research projects each investigate one or more sub problems from different perspectives and in different environments.

An important aspect in all our research efforts is the modular nature of software – we investigate dynamism and runtime change issues on the level of software components or modules. Consequently, our work has relevance to the domain of component configuration management. As discussed in [4] and as we have experienced, the emphasis in SCM has moved from development time to run time, from source code management to the integration and version management of the components, from management of implementation libraries to management of interfaces. The possibility of runtime change supports valuable system agility, flexibility, and adaptability, but it also gives rise to a number of problems, some of which can be alleviated by applying SCM. Important examples are dependence management and ensuring consistency. The SCM work on dependence management is good a starting point for future research, but it should be re-evaluated in the context of very frequent change and the crucial requirement of timely reaction. Similarly, the SCM mechanisms for version tracking and management have to be re-evaluated. With frequent runtime change it becomes also harder to achieve consistency, and SCM should research not only methods that try to ensure consistency before the change occurs, but also methods that enable successful system operation when the change causes an inconsistency. It is not possible to always precisely predict or test the effects of runtime change and therefore the runtime SCM mechanisms should have mechanisms to discover inconsistency and recover (e.g., rollback) from unsuccessful change. However, to achieve a consistent state a system might have to pass through intermediary inconsistent states and therefore mechanisms for transactional changes are also needed.

Service components can encapsulate not only software, but also hardware functionality. In this context, the issues like balancing limited underlying hardware and software resources during runtime and managing QoS become more important. Such issues are traditionally beyond the scope of SCM solutions. We believe that one of the main challenges for SCM in the future will be its integration with other management areas (fault, accounting, performance, and security management) and domains (device, desktop, network, system, enterprise, service management). Note that enterprise management software suites like Hewlett-Packard's OpenView, IBM's Tivoli, and Computer Associates Unicenter TNG already include some SCM functionality and integrate it with other management applications in the suite. However, we see the need for further work and research in this direction, at least on three tightly interrelated topics:

- Mapping SCM solutions to service management, i.e., to business-related issues and data. In order to better support this mapping, the price/performance ratio, uninterrupted service availability, and other issues (both technical and non-technical) relevant to end users must be addressed.
- Integration of SCM with system and network configuration management.
- Integration of configuration management and other management areas.

The previously given example of dynamic reconfiguration also illustrates these issues, as will be briefly discussed next. First, this dynamic reconfiguration problem requires integration of configuration management and fault (or performance) management. Further, the fault (or performance problem) in question can be software-based, but it can also be caused by an underlying computer hardware or network infrastructure, so the integration of application management with systems and network management is needed. Finally, as the ultimate goal of management is to minimize impacts on end users, this is also a service management problem.

We also believe that the importance of the issues of flexibility, availability, scalability, and performance of management solutions (including SCM) will further increase. Although our research is more focused on increasing flexibility and availability, we also examine scalability and performance issues in all our research projects.

Several recent industrial initiatives, like Microsoft .NET [7] and Sun Open Net Environment (Sun ONE) [10], are based on the concept of a Web service. Our research is compatible with these industrial initiatives and explores issues that they currently do not address. A Web service is a service component (in our definition) that communicates by means of XML-based standards. These initiatives show the trend towards service-based software systems, where monolithic applications are decomposed into distributed service components, and towards the “software is a service” business model, where software functionality is a service available via a network and not anymore a product that is deployed to consumers. This new situation in the software market will open a number of new challenges for SCM, some of which are already uncovered by our work.

REFERENCES

1. Feng, N., Ao, G., White,T., and Pagurek, B. Software Hot-swapping Technology Design. *Technical Report*, Systems and Computer Engineering, Carleton University, Ottawa, Canada, June 1999.

2. Feng, N., Ao, G., White, T., Pagurek, B. Dynamic Evolution of Network Management Software by Software Hot-Swapping. Accepted for *the Seventh IFIP/IEEE International Symposium on Integrated Network Management - IM 2001* (Seattle, Washington, USA, May 14-18, 2001).

3. Kniesel, G. Type-Safe Delegation for Run-Time Component Adaptation. In R. Guerraoui (Ed.) *Proc. of the 13th European Conference on Object-Oriented Programming - ECOOP ’99* (Lisbon, Portugal, June 1999), Springer.

4. Larsson, M., Crnkovic, I. New Challenges for Configuration Management. In *Proc. of System Configuration Management - SCM-9* (Toulouse, France, August 1999), Springer.

5. Mennie, D., Pagurek, B. An Architecture to Support Dynamic Composition of Service Components. Presented at *the Fifth International Workshop on Component-Oriented Programming – WCOP 2000*, held in conjunction with ECOOP 2000 (Sophia Antipolis, France, June 2000). On-line at: http://www.ipd.hk-r.se/bosch/WCOP2000/submissions/mennie.pdf

6. Mennie, D. W., An Architecture to Support Dynamic Composition of Service Components and Its Applicability to Internet Security. *M.Eng. thesis*, Carleton University, Ottawa, Canada, October 2000.

7. Microsoft .NET Web Site. On-line at: http://www.microsoft.com/net/

8. Oreizy, P., Medvidovic, N., Taylor, R. N. Architecture-Based Software Runtime Evolution. In *Proc. of the International Conference on Software Engineering 1998 - ICSE'98* (Kyoto, Japan, April 1998), pp. 177-186.

9. Oreizy, P. Issues in Modeling and Analyzing Dynamic Software Architectures. In *Proc. of the International Workshop on the Role of Software Architecture in Testing and Analysis* (Marsala, Italy, June/July 1998).

10. Sun Open Net Environment (Sun ONE) Web Site. On-line at: http://www.sun.com/software/sunone/