2010 IEEE Asia-Pacific Services Computing Conference

A framework for managing dynamic service-oriented component architectures

Walter Rudametkin[1,2] , Lionel Touseau[1], Didier Donsez[1], Francois Exertier[2]

1 Laboratoire d'Informatique de Grenoble
Grenoble, France
{Name.Lastname}@imag.fr

2 Bull SAS
Echirolles, France
{Name.Lastname}@bull.net

Abstract—**Software development is moving from monolithic to modular, dynamically composable applications. Modularity and dynamicity are the basis for software evolution since they provide the means of adapting and updating an application. Currently, service-oriented component models are one of the most advanced technologies for creating dynamic applications. These component models, which inherit concepts from both component-based software engineering and service oriented computing, provide a programming model that both supports and encourages dynamic reconfigurations. Although reconfigurations are possible, it is still difficult to manage a dynamic application's architecture, especially in highly dynamic environments. In this paper, we provide an overview of the benefits of service oriented component models and the main concepts used in their implementations. We provide a model that reifies important concepts and can be used to manage the application's architecture and its dynamic reconfigurations. Finally we propose a generic framework that allows for the creation of specialized architecture managers, capable of both monitoring and controlling dynamic service-oriented component applications.**

Keywords-Service oriented components, dynamic reconfiguration, adaptive systems.

I. Introduction

Increasingly, software is required and designed to accommodate new features after the design and deployment stages. Software needs to dynamically adapt and evolve at run time to minimize interruption and, if possible, never stop running. We call this *software evolution* [1,2]. Software evolution's goal is to allow applications to adapt to changing requirements, correct themselves in case of bugs, be updated or patched, change their architecture, adapt resource consumption, or dynamically reconfigure for any other number of reasons. These needs are not new and have been addressed by numerous researchers over the years [3], but given new techniques and design principles, dynamic reconfiguration is becoming more popular. Among the many approaches to achieve dynamic reconfiguration we can mention component-oriented programming (COP) [4], service-oriented computing (SOC) [5], and service-oriented components [6], which are the SOC principles applied to component models for greater flexibility.

Following this tendency through, future applications will become highly dynamic [20], they will require ever more rapid adaptability to better meet their goals with reduced delay. Also, correct operation is consistently required no matter the changing conditions, be it external changes in the environment of the system (e.g., arrival of new devices, new legal regulation, market opportunities), or be it internal changes in the application itself (e.g., software errors, security patches, resource consumption), adding to the burden of managing dynamic applications.

Highly dynamic environments create new problems when managing applications. Many operations require automation because of their low-level quickly changing details and the time constraints involved. Higher-level abstractions are needed to better understand the application in its entirety, and configurable policies are required to ease administration. Because non-functional requirements will undoubtedly vary, goals and objectives at a per-system level are required to drive architecture management strategies. This means that there is no *"one size fits all"* solution to the problem, but many solutions according to the current needs of the application at hand.

An application's software architecture is its internal structure defined by the different modular units that are composed to provide functionality. In the case of service-oriented components, the architecture is the component instances that are interconnected to form the application. These component instances have dependencies that must be met before execution is possible. As we will explain, there are different types of dependencies which need to be treated differently. Components also have lifecycles that affect the architecture and must also be considered. Component granularity is also important, because the tendency is to have many smaller components to increase flexibility and adaptability, but this complicates administration because the number of bindings and dependencies increases. Furthermore, updating and adapting software requires handling un-deployed or previously unknown components. These components are generally stored in external repositories which provide necessary metadata.

In this paper we characterize dynamic reconfigurations in service-oriented component models and provide a framework which can be used as the basis for constructing architecture managers that can control (e.g., perform reconfigurations) and monitor the application. In sections II and III we introduce service oriented computing and service oriented components. We give an overview of dependencies and lifecycles in section IV. In section V we present our approach and our framework, while section VI presents our implementation. Section VII details our use case: the impact of dynamic reconfigurations. Finally, sections VIII and IX provide related work and conclusions respectively.

978-0-7695-4305-5/10 $25.00 © 2010 IEEE
DOI 10.1109/APSCC.2010.99

IEEE computer society

II. Service-oriented computing

Service-oriented computing (SOC) [5,7] is a paradigm that defines a service as the fundamental unit for application design. Services are self-describing components that support composition of distributed, and more recently centralized, modular applications. Among the objectives of SOC is to define and reduce dependencies between functional units and to promote substitutability. By reducing dependencies, each element can evolve separately making the application more flexible. SOC is based on three actors:

- A service provider offers a service.
- A service consumer uses a service.
- A service registry contains references to services.

Services are described using a service specification, which is a description of its functionality (i.e., a service interface), and which may include its non-functional characteristics and semantics. A service provider publishes its service specification and the reference to the service implementation using the service registry. Consumers may search for services using the registry and then invoke them once they have a reference to the implementation. This provides discovery, selection, binding and composition of services. In figure 1 we can see the basic architecture.

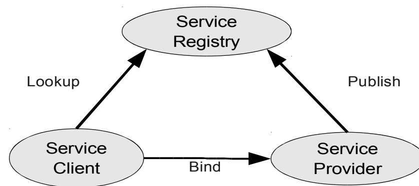

Figure 1. The basic Service Oriented Architecture [5]

SOC provides characteristics that are exploited for dynamic applications (applications that adapt at runtime). In general, they provide the means to achieve substitutability, which is the basis for dynamism. We list them next:

- *Loose coupling:* a consumer needs only to know what is specified in the service specification.
- *Late binding:* a consumer may consult the registry at any time to bind to a service implementation.
- *Dynamic resilience:* service consumers do not rely on the same service implementation being returned.
- *Location transparency:* providers and consumers are oblivious to the underlying infrastructure.

In order to build complex applications it is necessary to compose services to provide higher-level services. Service providers may require other services in order to operate correctly. This entails service-dependencies, where providers publish their services when their dependencies are met, and they may retreat them when not.

Service-oriented applications require additional attention and may be difficult to implement, intensive and error-prone. The complexity involved has lead to component-based approaches that use the SOC concepts but advocate the separation-of-concerns principles. The next section describes how SOC concepts are merged into component models to provide dynamically adaptable software systems.

III. Service-Oriented Components

A component is a software package that encapsulates a set of functions or data. Components can be seen as black-boxes whose functionality is expressed by clearly defined interfaces [4]. These interfaces are used to connect components for communication and to compose them to provide higher-level functions. The interface acts as the signature for the component, consumers need only know the interface and can be naive of its implementation. Cervantes [6] presented the general principles of the service-oriented component model, an SOA extension to component based development. The proposed principles are the following:

- A service is provided functionality.
- A service is characterized by a service specification which describes its syntax, behavior, and semantics.
- Components implement service specifications.
- The service-oriented interaction pattern is used to resolve service dependencies at run time.
- Compositions are described using specifications.
- Service specifications provide the basis for substitutability.

The model that results from these principles promotes service substitutability because compositions and dependencies are expressed in terms of specifications. This makes it possible to develop constituent services independently as well as have variant interchangeable implementations. As in SOA, locality is largely irrelevant. In centralized implementations (i.e., single memory space), a component may provide a service but internally act as a proxy, transparently providing distribution. In traditional component-oriented models, selection occurs at design time, when bindings and the architecture are specified [10] (some models do provide run time adaptation [8,9]). The selection process for service-oriented components occurs at run-time. Component instances are created by the execution environment and the application starts when the main component's dependencies are satisfied. The service oriented component model is thus flexible and powerful.

A. *Abstraction levels*

Implementations of service oriented component models vary but require an underlying dynamic framework to provide them with the necessary mechanisms for run-time adaptation. Dynamicity relies on the service-oriented computing paradigm (i.e., consumer, provider, registry, service specification) to provide substitutability. Depending on the specific technology, concept mappings may vary, but for this work we provide an overview of implementations using the object oriented paradigm.

Deployment unit or module: is used for installing, updating and removing components. A deployment unit provides component types (and other resources) and contains metadata related to dependencies and features.

Component type: is the component specification. It defines the implementation of services and the component's dependencies (by means of service specification dependencies). Because it implements services it is used to

satisfy other component's service dependencies.

Component instances: these are the run-time entities that are composed during execution. A single component-type may be instantiated many times. Components are bound (i.e., bindings) in order to communicate (i.e., invoke services), letting them perform calculations, share data, etc.

B. *Mapping components to objects*

Service-oriented component-models are usually written in object-oriented languages (although some do exist in other paradigms). Components are not natively supported by many platforms, so they may be more or less transparent depending on the underlying framework, the abstractions, and development model. It is important to visualize component-to-object mappings to better understand the dependencies that exist, which go further than clear-cut service specifications. These mappings become ever more interesting in centralized component models, because they show datatypes and service references that are shared among component instances. There are two concepts we are interested in that affect dependencies, class and object:

Class definitions: are the basic unit of design in object oriented programming. They specify attributes and methods, which make them a mix of data and behavior in an encapsulated entity. A developer, even when creating components, writes classes. Elements from the component model, including the component's business functions, the actors (i.e., consumer, provider, registry), services, specifications, datatypes (including in the specification), are mapped to their implementations in the object oriented language. The execution framework does not distinguish between a type of object that represents a component, service or datatype, they all consist of the same abstraction.

Object instances: are the instantiation of classes. These run-time entities hold the state of the application. There is no mapping that tells us that an object belongs to a specific component instance or component type since these abstractions are generally not reified by the framework.

In figure 2 we show the abstraction levels that exist in service oriented component frameworks at run time, along with their implementation mappings to the object oriented paradigm. The deployment and design levels show higher abstractions and are the views a user will work with. At the deployment-level one sees modules (i.e., deployment units) on the framework and can manipulate their lifecycles, including installation and removal, which are the two basic primitives. Modules contain component types (i.e., component definitions), which are instantiated by the framework to create component instances. Component types and component instances are also commonly reified when a user requires more details at run time. Component types are in fact a set of class definitions. At the class level, classes inside modules may reference classes from other modules. This is common, for example, for datatypes which are specified in the service specification and shared. These cross-references of classes exist precisely because of data and implementation sharing. At the run-time level, we show object instances and how they reference objects that are defined by classes in different modules. References can be entangled between modules even when we follow a service oriented computing approach that promotes loose-coupling. As a note, we provide an outlined module in the figure (dotted rectangle) as reference to where the elements came from, but this abstraction is not reified beyond the deployment view.

IV. Dependencies and Lifecycles

Dependencies are the primary constraint to performing dynamic reconfigurations. Missing dependencies affect the lifecycle of components. In our model we show that

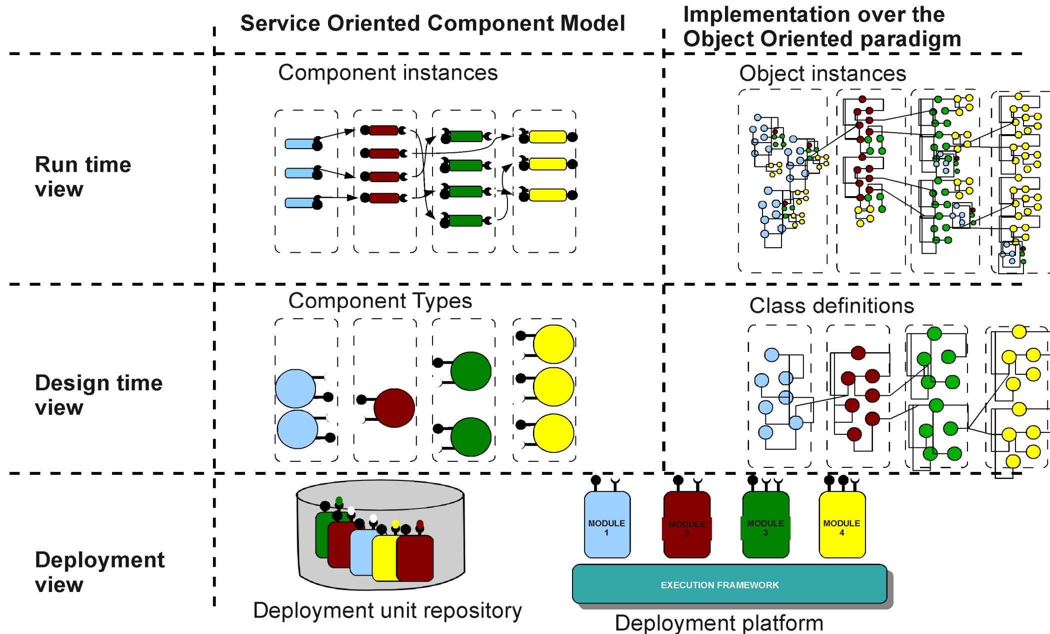

Figure 2. Abstraction levels in service oriented component model implementations.

implementation code is provided by modules in the form of component types, and that the granularity of updates is the module[1]. Changing the architecture at a finer grain (that of component instances) is possible, but since no new implementation code is provided at other levels, the changes are limited to creating (or destroying) new instances and changing bindings. This leads us to evaluate service oriented component dependencies at two basic levels, implementation dependencies, which are static, and service dependencies, which are dynamic.

A. Dependency types

Static dependencies are those where a reconfiguration requires restarting and reinitialization the module, causing its full state to be lost and all instances of its components to be destroyed. Because the unit of deployment is a module, and at the module level is where implementation dependencies are handled, the module is the granularity that is directly affected. State-loss and instance destruction are required when a module imports implementation code from another, and the provider module changes. For example, if module A requires classes from module B, and B updates, we must also update A[2] to use the newer implementation. This type of dependency is common for datatypes specified in service specifications and for modules that provide libraries. Implementation dependencies are always mandatory for a module to operate correctly and are costly because they cause the destruction of dependent modules' component instances (and states) when changes are applied.

Dynamic dependencies are those where a reconfiguration is possible without restarting the module and loosing state. These dependencies occur at the service level and are the direct benefit of the principles of service oriented computing. Provided and required services may both change dynamically. A component may provide a service through the registry at one moment, and remove it at another. Also, a required service may be optional, giving the component degraded functionality if not available. Dynamic dependencies affect the component instance and cause rebinding to a compatible service if a change occurs. If no compatible dependency is found, then the component instance is stopped, and its provided services removed, until its dependencies can be resolved.

Other types of dependencies, which we call *resource dependencies*, generally regard configuration, and may be either static or dynamic. For example, a communication port, according to how the component is implemented may either be static, and require reinitialization of the module to change, or may be dynamic having the component internally handle the change. Also, a port may not be used by two components, so these dependencies help avoid conflicts at run time. Other examples include hardware devices and files. In general, these dependencies specify if the resource they require can be shared or not (e.g., a file might be read simultaneously) and if the dependency is static or dynamic. The effects at run time are the same as for static dependencies if the resource is static, and dynamic dependencies if the resource is dynamic.

In figure 3 we present the dependencies that a module may require and provide. In order to keep our model as simple as possible, we hide the fact that internal component instances are the actual implementations providing and requiring dependencies. This is important when constructing our dependency graph, because the amount of component instances that will be created at run time is difficult to know beforehand unless a static creation policy is used. In the figure, one can note that dependencies are of three types, implementation, service and resource.

B. Module and component lifecycles

A lifecycle is the ensemble of states that an entity may be in, of which it is in only one at any given moment. In our case, we are interested in the lifecycles of component instances and modules, both at run time, and we will show how they are intertwined. Figures 4(a) and 4(b) show the individual lifecycles of both, components and modules. These states illustrate dependency constraints at run time. For example, regarding modules, if all static dependencies are resolved, a module is valid and provides its exported implementation (and resources) to other modules. If all dependencies are not solved or are no longer solved, the exported items are no longer provided to other modules. Regarding components, if all services are available, then the component is started and its state is set to running, providing its services. If mandatory services are not available or become unavailable, the component is halted and its services removed. It is important to remember that component types are delivered in modules, and so the lifecycles of component instances' are directly impacted by that of their underlying modules. In figure 4(c) we give an overview of what this looks like. When the module is invalid, components are destroyed. When the module is valid, that is, its dependencies satisfied, components may now resolve their own dependencies in order to run. As a general note, dependencies directly impact the lifecycle of modules and components, and thus, impact the elements that they provide like implementation code and services.

V. A GENERIC FRAMEWORK FOR ARCHITECTURE MANAGEMENT

As seen in the previous sections, the required technical knowledge to manage service oriented component applications efficiently is extensive. Furthermore, let us not forget that in highly dynamic contexts, the need to react, and react quickly, is essential in order to meet ever changing requirements. Given all the low-level details one must consider, the creation of architecture managers is a difficult task. For these reasons, automation is required.

In order to avoid falling into the trap of a single monolithic manager that satisfies all situations but is not flexible nor optimized, we advocate the need to develop many smaller, simpler managers that are specialized to the

1 It is possible to create a module that is composed of only one component, merging the deployment unit and component type concepts.

2 Module A may continue to operate with the old implementation dependencies but it would be necessary for it not to communicate with modules that rely on the newer version because low-level incompatibilities may occur among classes.

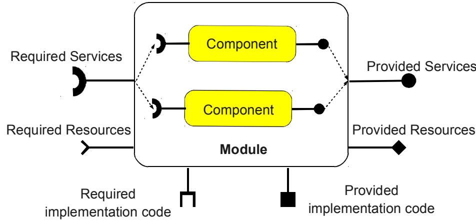

Figure 3. Module dependencies

goals of the application at a given moment. These managers should be interchangeable when needed, complementary if desired, optimized for a certain purpose, quicker to implement, and thus give way to greater adaptability. We feel that no single manager is sufficient for all needs, and that even the architecture manager for a single application may require being changed over its lifetime. In general, different objectives require different strategies, and different strategies require different architecture managers.

Our solution to the problem of administrating service oriented component applications is to create a framework that provides necessary and reusable services for the task of implementing architecture managers. As explained in earlier sections, managing the architecture of service oriented component applications is a delicate balance between managing dependencies (static and dynamic), granularity (size and quantity of components) and lifecycles. Our framework provides abstractions in a central and unified manner. In figure 5 we present the architecture manager framework, including the basic components that provide essential services to the architecture managers themselves. These services provide the means to monitor the application, to control and change it, to resolve and deploy dependencies, and to communicate with other distributed services. In the figure, we also give an overview of the larger context and show how our framework and the application that we are managing interact with the different systems. We will explain the base services next.

A. *Overview of the architecture manager framework*

The architecture manager framework provides the foundation for implementing architecture managers. The services we have created are the fundamental building blocks to ease the development of such managers. These services provide different functions, such as monitoring, deploying, control, communication and dependency management. We shall present them in more detail.

1) Run-time manager: this is the essential block used for monitoring and controlling the application at hand. It's either co-located with the application or is fundamentally integrated into the dynamic framework the application is running on (or a mix of both). It is precisely this manager that reifies the state of the application and of the dynamic framework regarding dependencies, services, components, modules and other abstractions.

2) Resource manager: it is also co-located with the application or integrated into the dynamic framework and provides information regarding the resources available on the system and their levels of consumption. This manager is necessary in order for the architecture manager to consider opportune moments for dynamic reconfiguration. For example, when the CPU and network usages are low, and there is enough disk space, the architecture manager may choose to pre-deploy modules to the system to avoid contention at peak periods.

3) Resolver: is essential in order to verify that the dependencies of a dynamic reconfiguration will be met. It analyzes a number of modules and component types, and may also autocomplete missing dependencies using either repositories or distant services, or both. Without this the architecture manager cannot be sure that what it will deploy (or undeploy) will be able to run (or keep running), or what the state of the application will be if a dynamic reconfiguration is to be executed. It is also necessary for choosing the reconfiguration actions necessary should an unforeseen event occur and leave the application in an invalid state. After the architecture manager, it is the most likely component to require re-implementation since many objectives of the architecture manager must be configurable in the resolver (e.g., find the minimum number of modules to satisfy dependencies, module policies and preferences).

4) Distant services manager: this component is in charge of discovering distributed frameworks and services that are available for use by the application. These services may be used to satisfy dependencies of the application, and should be considered by the resolver. The distant service manager also handles communication, preferably transparently, since

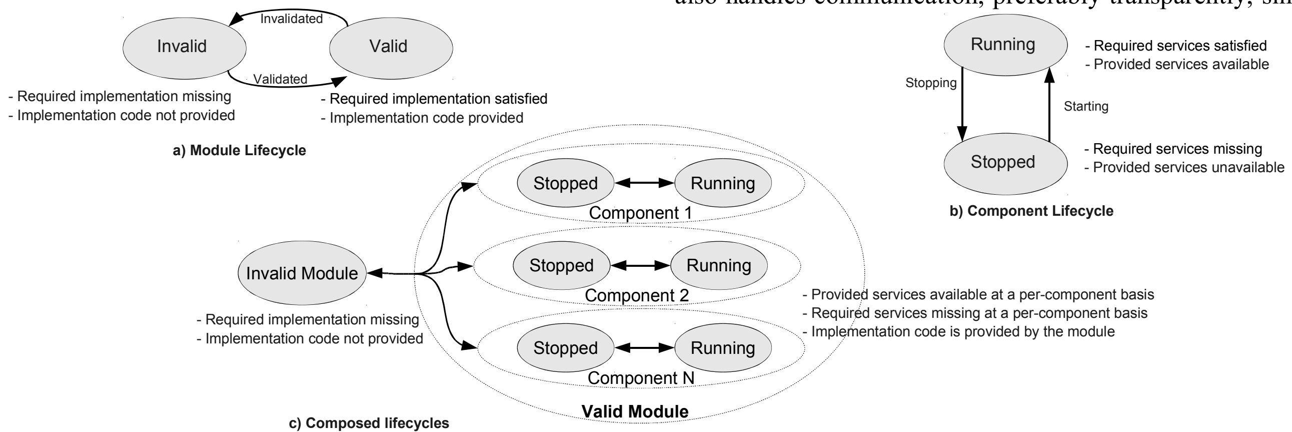

Figure 4. Lifecycle of modules and components, separately in a) and b), then composed in c).

the service bindings must performed. If transparent distribution is not possible, then the use of this service becomes limited since it would require the service oriented components themselves to be distribution aware.

5) Repository manager: handles deployment units and their metadata. It provides the necessary information to the architecture manager and the resolver in order for them to perform dynamic reconfigurations.

6) Architecture manager: the architecture manager the implementation of the objectives and strategies regarding application management. This component is specific according to the goals of the application. It uses the other components in the framework in order to achieve those goals. Most of the intelligence regarding administration is located here.

VI. Implementation of the framework

In order to demonstrate our approach, our framework is both implemented using, and administers applications for, the OSGi service platform [13] and the iPOJO component model [14]. OSGi is a platform that provides dynamic deployment of modules which are called bundles. iPOJO is a service oriented component model built on top of OSGi providing many abstractions and separating the code related to dynamic behavior from functional code. Part of our run time manager must be co-located with the application and uses hooks into the OSGi platform to detect the arrival of new modules and services, and to detect the import and export of implementation classes which are declared in the form of Java packages. iPOJO provides information regarding component instances, component types and bindings. We have also added metadata to bundles, particularly concerning resource dependencies, because they are not natively expressed in OSGi metadata. Our resource manager is also partially co-located with the application, and uses the SIGAR library [24] to obtain the resource usage of the dynamic framework and system. Regarding deployment, various mechanisms already exist for storing bundles (modules), accessing their metadata and for deploying them to a running OSGi platform. In our case, we have opted for using OBR [25], although we feel it currently does not meet all our needs. The distant services manager is a simple component that delegates to ROSE[26], which uses a proxy system to communicate transparently with distant services. The resolver, one of the more crucial services that must be provided, currently uses the JBOSS OSGi Resolver, but plan on including P2 [18], the resolver used in Eclipse, since it provides more functionality. Finally, using the available services that we defined, we have implemented an architecture management component for a specific use case that we present in the next section. It should be noted, that while certain services require partial co-location, we provide a communication mechanism, using the XMPP protocol, to send events and actions to and from the dynamic framework under management. Thanks to the OSGi common base, we have the liberty of co-locating all services with the application, providing rapid analysis of events but also disturbing the application, or, distributing them increasing latency but also improving isolation.

VII. Use cases

There are many use cases where an architecture manager for a service-oriented component application is useful. To mention a few, a manager for minimizing application downtime by ensuring dependencies are always met and pre-calculating replacement dependencies, should a dynamic reconfiguration be necessary. Another use is to optimize a specific resource, such as CPU consumption (given the modules are annotated with such non-functional metadata), thus deploying the modules that implement the required functionality (i.e., service specification) and minimize usage of the resource. We have taken a particular interest in the impact of dynamic reconfigurations and have implemented an architecture manager for impact analysis to

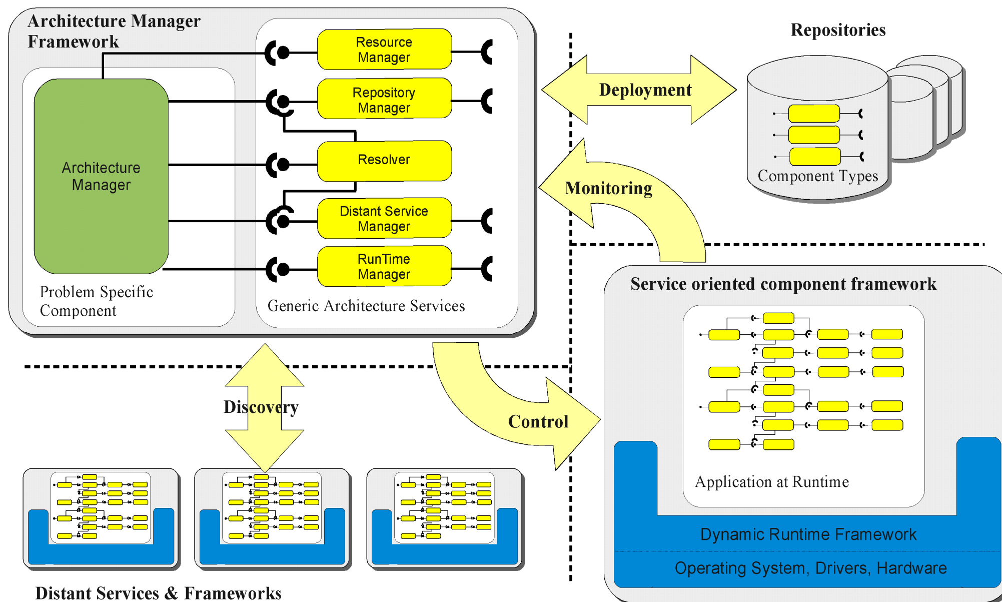

Figure 5. Conceptual model

validate our approach, which we describe next.

A. The impact of dynamic reconfigurations

Dynamic reconfigurations, even following the principles of service oriented computing, may be costly and have a stronger impact than one might consider if only looking at service dependencies. In the previous sections we presented the different types of dependencies (section IV.A) and how they affect the lifecycle of modules and components at run time. When changes to a dependency occur, the impact on dependents can be that of total state-loss for the dependent, in the case of static dependencies, or simply rebinding to another service in the case of dynamic dependencies. Our architecture manager attempts to calculate this impact. We create a model of the run time dependencies, before reconfigurations are applied. We then perform the reconfiguration to the in-memory model in order to see the effects beforehand. Reconfigurations are expressed at high abstraction levels (e.g., the module level), simplifying the task of deployment and configuration, and sent to the architecture manager. (Our method does not inhibit using low-level reconfigurations, e.g., individual component instances, but we believe that higher abstractions simplify platform administration.) For example, an administrator decides to update a module. Our architecture manager calculates the impact on dependencies at different levels across the application and according to dependency types, and responds with a list of modules, component types and component instances that will be stopped, started, restarted installed, uninstalled, changed bindings, state-loss, and so forth. In order to do this, our in memory model is actually a dependency graph, where nodes are component instances (nodes also reference their parent module). Edges between nodes are colored and labeled in order to distinguish both the dependency (label) and the dependency type (color). For example, the label of a service-dependency specifies the service specification in question (e.g., printer-service), while the color (e.g., red) specifies a dynamic dependency.

It is important that all dependencies be reified for them to be evaluated. Resource dependencies, such as ports, are a special case and may cause conflicts between components. To complete the graph with these dependencies, a new node is created to represent the resource (e.g., port 8080) and an edge of the type of dependency (e.g., static + non-shareable) is created. The architecture manager verifies the edges to these nodes to avoid conflicts for static or non-shareable.

Regarding impact, for each reconfiguration that is applied to the model, we take calculate the effect on existing components and modules. There are two types of impact: *Dynamic,* which affects component instances. If a dynamic dependency exists to a component that changes, the dependent component will require a rebind to find another suitable service (or wait for the same service to return). If no suitable service is found, the component is stopped. In either case, the impact is to rebind or stop, and it is made explicit. An impacted component may affect other dependent components, causing them to stop or rebind, and so on, in a domino effect. *Static dependencies* affect both modules and components, because components' lifecycles depend on those of modules'. If an implementation dependency changes, all component instances of the dependent modules are destroyed causing state-loss. This affects dependent modules and components and the effects must be propagated. Basically, all statically dependent modules are restarted in a similar domino effect.

The total cost of a dynamic reconfiguration is the sum of the affected modules, the component instances lost (state-loss), the (re)creation of component instances, the number of re-bindings, the modules installed, the modules restarted and the components and modules that do not start because of unsatisfied dependencies. If non-functional component metadata were available, the manager could calculate the total resources used in a dynamic reconfiguration (e.g., memory, cpu, disk) by adding the individual resource consumption for each reconfigured element. At the moment, we are limited to the network consumption and disk space usage, since the size of bundles is easily obtainable.

An additional element we have added is to automatically satisfy missing dependencies, in order to minimize downtime. In our case, any dependency that is calculated to be missing in the application after a proposed reconfiguration, is satisfied by deploying the necessary modules or, if possible, by binding to a distant service.

We have tested our tool on the JOnAS application server which uses OSGi at its core [15]. With a common install of around 150 bundles and a highly complicated dependency graph centered around a number of crucial bundles, we note that the impact of dynamic reconfigurations is costly.

VIII. Related work

Most work relating to dynamic reconfiguration in centralized software has focused on the reconfiguration mechanisms themselves, and is visible in component models such as Fractal [8], Koala [9] and OSGi [13]. These component models tend to simplify reconfiguration thanks to mechanisms introduced into the component membrane, such as proxies that detect quiescent states (e.g., Fractal), or the use of specific API, development models and architecture approaches, as is the case for OSGi. Other work, as presented by Oreizy and Taylor [16,17] present software adaptation and compare architectural styles for run time adaptation, but focus on distributed applications and do not look at component models or centralized mechanisms. In our work, we provide an framework for the construction of architecture managers and provide a specific architecture manager that analyzes the impact of dynamic reconfigurations for (mainly) centralized, service oriented component frameworks.

Related to OSGi we can mention P2 [18], an advanced dependency management and module distribution system provided for the Eclipse Framework. This work focuses on resolving constraints and dependencies, but does not focus on the architecture administration necessary on the platform itself. Further, the impact of these deployments is not calculated. Other work that is similar to this is that of Touseau [11,12] which seeks to limit the impact of reappearing services by means of SLA mechanisms. There

is also work being done to ensure that metadata to describe modules and components is correct by means of static analysis [19]. We find this work to be necessary to ensure dynamic reconfigurations, although for the time being our approach supposes metadata to always be correct.

Finally, we feel our work has brought us close to the domain of autonomic computing. The general purpose of Autonomic Computing is to enable self-management of software systems and to minimize human intervention [21]. Our work is similar in this aspect, because we attempt to increase automation of architecture management by providing a foundation framework from where to start. One difference is we focus on low-level details, such as implementation dependencies in service oriented component frameworks, and not on high-level strategies, since the underlying framework for implementing those strategies does not exist for service oriented component platforms. Rainbow [23] is a similar framework conceptually; but focuses on distributed systems. Also, the work done in CEYLON [22] is interesting in that they compose smaller, simpler, reusable strategies, to obtain otherwise complicated autonomic managers. Implementing a composing mechanism for simple architecture managers in our framework would certainly be interesting, but we have not worked out how possible conflicts between managers would need to be solved, nor if this is wise solution due to added resource consumption and complexity of the manager. Furthermore, we have not found comparable work regarding our use case, the impact of dynamic reconfigurations in these types of frameworks.

IX. Conclusion

In this paper we have proposed a generic framework for the construction of architecture managers. We propose that architecture managers should be simple, optimized for specific goals, and interchangeable. (Different problems most likely require different solutions, even if the building blocks are the same.) We have created an architecture manager capable of analyzing the impact of dynamic reconfigurations, before or after they take place, and of resolving missing dependencies to keep the application running as “best as possible”. We have also specified different types of dependencies that exist in modern service oriented component models and how they affect otherwise loosely coupled components at run time and why they need to be properly considered in order to coherently manage an application's architecture. Regarding our use case, we show that it is necessary to have an understanding of the impact of dynamic reconfigurations because of their hidden and implicit cost, especially considering the domino effect that takes place across the application. In the future, we plan on using our current framework to construct an architecture administrator capable of evaluating the correctness of dynamic reconfigurations and which provides mechanisms and strategies to minimize the impact to localized areas of the architecture (e.g., temporary service substitution, reconfiguration at off-peak periods). We will continue to focus our work on highly dynamic environments, where consumers and providers may continually appear and disappear without rapidly and without notice.

References

[1] Lorcan Coyle et al., “Guest Editors' Introduction: Evolving Critical Systems”, Computer, vol. 43, no. 5, pp. 28-33, May 2010

[2] Bernard Cohen, Philip Boxer, “Why Critical Systems Need Help to Evolve”, Computer, vol. 43, no. 5, pp. 56-63, May 2010

[3] Peyman Oreizy et al., “Architecture-based runtime software evolution”, ICSE, p.177-186, April 19-25, 1998, Kyoto, Japan

[4] C. Szyperski, Component Software: “Beyond Object-Oriented Programming”, Addison-Wesley Longman Publishing, 2002, p. 448.

[5] M. P. Papazoglou , D. Georgakopoulos, “Introduction to Service Oriented Computing”, ACM, v.46 n.10, October 2003

[6] H. Cervantes and R.S. Hall, “Autonomous adaptation to dynamic availability using a service-oriented component model”, ICSE, 2004

[7] M.N. Huhns and M.P. Singh, “Service-Oriented Computing: Key Concepts and Principles,” IEEE Internet Computing, 2005

[8] E. Bruneton et al., “The FRACTAL component model and its support in Java”, SPE, v.36 n.11-12, p.1257-1284, September 2006

[9] R. Van Ommering, et al., “The Koala component model for consumer electronics software,” Computer, vol. 33, no. 3, 2000

[10] Marino, J. and Rowley, M. 2009 Understanding SCA (Service Component Architecture). 1st. Addison-Wesley Professional.

[11] L. Touseau et al.,: “Towards a SLA-based Approach to Handle Service Disruptions”, IEEE SCC (1) 2008: 415-422

[12] Lionel Touseau, PhD dissertation “Politique de liaison aux services intermittents dirigée par les accords de niveau de service”, Université Joseph Fourier, Grenoble, may 2010

[13] OSGi Service Platform Core Spec, 2005, http://osgi.org.

[14] C. Escoffier and R.S. Hall, “Dynamically Adaptable Applications with iPOJO Service Components”, Software Composition, 2007.

[15] M. Desertot, et al., “A Dynamic Service-Oriented Implementation for Java EE Servers”,IEEE Conference on Services Computing, 2006

[16] P. Oreizy et al., "Runtime software adaptation: framework, approaches, and styles," ICSE, ACM, 2008.

[17] Richard N. Taylor et al., "Architectural Styles for Runtime Software Adaptation", Conference on Software Architecture 2009

[18] D. Le Berre, P. Rapicaul, “Dependency management for the eclipse ecosystem: eclipse p2, metadata and resolution”, IWOCE, 2009

[19] J. Bauml, P. Brada, “Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee”. EUROMICRO-SEAA 2009: 428-435

[20] E. D. Nitto, C. Ghezzi, A. Metzger, M. P. Papazoglou, and K. Pohl, "A journey to highly dynamic, self-adaptive service-based applications," Automated Software Engineering, 2008.

[21] D. M. Kephart, Jeffrey O. et Chess. The vision of autonomic computing. Computer, 36, 2003.

[22] Y. Maurel et al., "CEYLON: A Service-Oriented Framework for Building Autonomic Managers," EASE, 2010, pp. 3-11.

[23] A. Huang, D. Garlan, and B. Schmerl, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," International Conference on Autonomic Computing, 2004, p. 276.

[24] http://support.hyperic.com/display/SIGAR/Home

[25] OSGi Alliance. RFC-0112 Bundle Repository, February 2006.

[26] http://wiki.chameleon.ow2.org/xwiki/bin/view/Main/Rose