A Model-Driven Approach for Dependable Software Systems

Michael Jiang
Motorola Labs, Motorola, Schaumburg, IL 60196, USA
michael.jiang@motorola.com

Zhihui Yang
Mobile Device, Motorola, Libertyville, IL 60092, USA
zhihui.yang@motorola.com

Abstract

High dependability is a key requirement for many types of systems, such as safety-critical systems, telecommunication systems, and mission-critical software systems. Although software components and web services are proven technologies to tackle design complexity, their reliability affects the reliability and availability of the systems they are part of. The composition of components and web services further complicates the issue. For highly dependable systems, the faults of components and web services have to be minimized to achieve overall system dependability.

This paper describes a model-driven engineering approach to improve the dependability of domain-specific software systems built with component and web service composition. In this framework, web services and components are specified as model elements and their dependability is enhanced by generating both functional code and protective mechanisms to reduce the impact of component and service failures. The applicability of this approach is demonstrated in our implementation and deployment of mobile services.

Keywords: Dependability, availability, reliability, Meta-model, web services, and software component.

1. Introduction

Various design methodologies and architectural paradigms have been proposed to tackle the complexity and reliability of large-scale software systems. Both software components and web services are often used to reduce design complexity. Software components and web services serve as large building blocks and the construction of large software systems is to integrate and assemble these software building blocks. They provide flexible and scaleable solutions for the design and integration of complex systems and applications [1, 2, 3].

The construction of a software system with component and web service composition brings new challenges to the dependability of the system. Software components, including commercial off-the-shelf (COTS) components, offer various degrees of reliability, depending on their functional complexity, implementation, and deployment environment. Their reliability may be unknown in design phase or vary significantly depending on the platforms they run on. The composition and interactions of components and web services also contribute to the overall system dependability and further complicate system analysis. While some of the components may be hosted in the same execution environment of their applications, the deployment of web services is independent of their consumers. Hence, the availability and reliability of a web service is beyond the control of its client applications. For systems that require high dependability, the failures of components and services must be handled to minimize their impacts to the overall availability and reliability of the systems.

For clarification, the term dependability means both system availability and reliability throughout this paper. In some literatures, dependability may be used as a collective term for reliability, availability, maintenance, safety, and security.

This paper takes an integrated model-driven engineering approach to model both the function and dependability of software systems. In this modeling framework, domain models are developed to capture the logic of domain applications. Software components and web services are represented as formal model constructs to enable the specification of their composition, interactions, and dependability. A model-based development environment is developed for the creation of domain-specific models and application code is generated from the domain models. The generated code implements both the functions of domain applications and the handling of component and web service failures to improve the overall system dependability. Section 2 gives a brief overview of modeling software components and web services. Section 3 describes the domain-specific modeling framework. Section 4 describes the application of the modeling framework for building reliable mobile services. Related work is described in section 5 and conclusion is given in section 6.

2. Component and web service models with dependability extensions

A component is a unit of software and is viewed as a black box providing specific functions. Component-based development takes the approach of integrating and assembling these prefabricated components to construct applications, as seen in many software applications built with Enterprise JavaBeans (EJB), Component Object Model (COM), Distributed Component Object Model (DCOM), and CORBA Component Model (CCM) [3].

A web service is a programmable interface implemented and exposed by an application to other applications via standardized web protocols. It is a loosely coupled application or a software function independent of underlying implementation languages, transport protocols, and deployment platforms. The interface and semantics of a web service are represented by its WSDL (Web Services Description Language) specification. It specifies the messages exchanged for service invocation, deployment details for locating the specified web service, and protocol bindings for transport. Web services can be viewed as a special type of software components with higher degree of decoupling.

We take a model-driven approach to integrate components and web services and represent them as model elements. Their functions and relationships with other modeling or programming constructs are

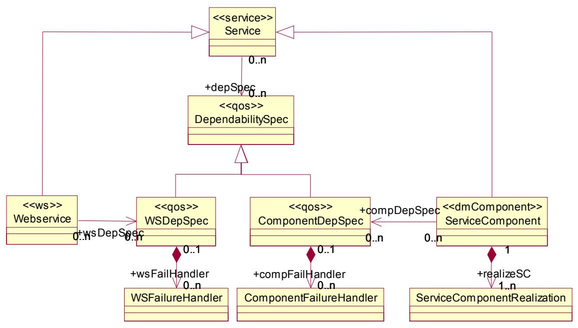

Fig.1. Modeling components and web services

specified at the model level. The modeling of components and web services is further extended with the specification of their dependability. Figure 1 shows the representation of components and web services as model elements and their associations with dependability specification. Both components and web services are specialization of class Service. Class ServiceComponentRealization specifies the Artifacts that provide concrete implementation for software components. For web services, the WSDL specification contains sufficient information for applications to locate the service providers and invoke services from the service providers.

Both components and web services are modeled and validated based on their interfaces for composition. Components and services are replaceable as long as they are compatible at the interface level. OCL constraints are also attached to the meta-models to capture more syntactic and semantic information for component composition, enabling stronger correctness checking beyond the syntax checking and semantic checking of the traditional component-based software development [5, 6]. The integration rules and constraints are enforced by the connectors linking components and services. With the formal specification of components and web services as model elements, their functional specification and interactions can be verified within the context of a domain model.

Class DependabilitySpec extends the specification of components and web services to incorporate the specification of component and service failures and the strategies to handle these failures. Different types of failures and their associated handling strategies are described in Class ComponentDepSpec and WSDepSpec. Failure types include both value-based and exception-based types. For value-based failures, the return values from the component and service invocation indicate the types of failures. For example, the return value of "-1" may be used to indicate a specific failure of component invocation, similar to the conventions used in some of the UNIX library calls. Exception-based failure types represent the types of exceptions declared in and thrown by components and web services.

Specific failures of components and web services and their handling strategies are specified in class ComponentDepSpec and WsDepSpec respectively. The following types of handlers are defined for components: re-invoking the component service, restarting the component, and ignoring component failure. The failures of web services are handled in similar fashion with the exception that there may be multiple service providers for a particular web service.

Class WsDepSpec defines a list of web service providers and allows the invocation of a web service from any available web service providers. In the event of a web service failure, the list of web service providers can be invoked sequentially until the invocation succeeds or the list depletes. Class DependabiltySpec also defines the strategies to control how each of the failure handlers are triggered. For example, the control mechanism may specify number of component failures before a component will be restarted. It may also specify how redundant web services are invoked in the event of web service failures.

One of the implications for such protection mechanisms is that it requires the protected software units to be stateless to ensure the failure handling is transparent to the application logic. Since software components and web services are generally defined to be stateless, their failures can be handled by the above protection mechanism without impacting the application logic. For stateful software units, the failure handling needs to be customized.

The modeling of components and web services along with their dependability specification is to automatically generate the protection mechanisms based on the handling strategies declared in the dependency specification. The dependability specification of a component or a web service is implemented by a delegation pattern with extension to handle the failures of component and service invocations. Invocations of components or web services are delegated to their concrete implementations and failures from invocations are intercepted, interpreted, and handled according to the strategies in the dependability specification.

3. Modeling framework for mobile services

This section describes the modeling of mobile services and applications. The structures, behaviors, configuration, and deployment of mobile services are specified using Eclipse Modeling Framework and our extensions to this modeling framework for the modeling, integration, validation, and runtime support of components and web services. A development environment based on the domain meta-models is developed for the specification of domain applications. Code is generated from model specifications for both the application logic and the failure protection mechanisms to handle failures of components and web services.

3.1. Mobility service models with dependability specification

Developing a mobility service model involves the specification of model constructs representing the domain elements and their relationships. Some of the key domain elements are the domain services, messages, message handlers, and service processors. Figure 2 shows a portion of the mobility service meta-models for message-based services and the dependability specification which components and web services are associated with. Class MobilityService is the generalization of all types of concrete mobile services, such as message-based services (MessagingService), location-based mobile services (LocationService), and mobile web services (MobilityWebService). Class MobilityService also functions as a container for deployment configuration, service processors, web services, and components interacting with legacy systems.

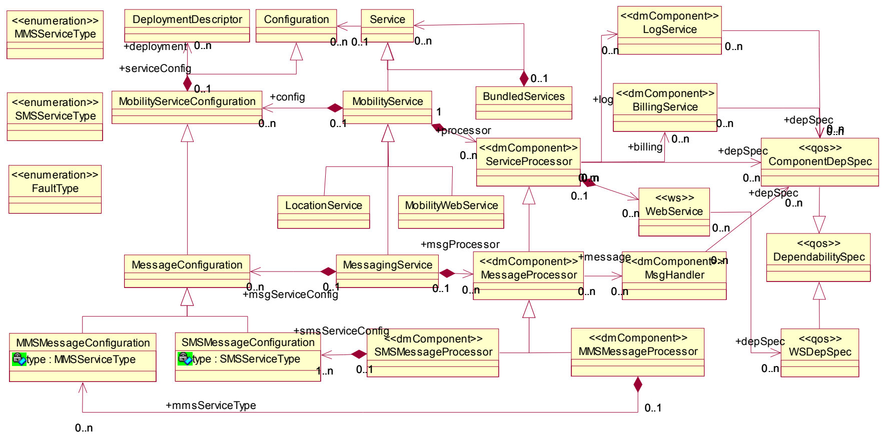

Fig.2. Portion of the domain meta-models for mobile services

Class MobilityServiceConfiguration specifies the various parameters and options to facilitate the generation of deployment descriptors and the packaging of applications for platform-specific implementations. Class MessageConfiguration carries deployment options specifically for message-based applications and services. SMSMessageConfiguration specifies options for SMS (Short Message Service) messages, with service type described by enumeration type "SMSServiceType".

The integration of components is accomplished through their interfaces discussed earlier. A component can be connected with another if the required interface type matches that of the provided interface. A component is also replaceable with another component if their interface types match, thus enabling reuse of components across different applications and services. The properties of components are used to specify further details of component implementation for code generation and application packaging.

Components and web services are associated with class DependabilitySpec to specify their failures and corresponding handling mechanisms. The dependability specification for class ServiceProcessor, LogService, and BillingService is specified by class ComponentDepSpec. Class WSDepSpec is associated web services for dependability specifications. As discussed in previous section, the types of failures and their associated handlers are captured by class DependabilitySpec. The dependability specification together with the functional specification of components and web services provides complete information to generate the protection code and associate it with the invocation of components and web services.

The explicit modeling of domain structures, rules, and constraints in the domain meta-models also facilitates the construction of tools for model-based development discussed next.

3.2. Generating dependable applications

The specification of a domain application and its domain meta-models are interpreted to generate code for the application. The code generated includes two related packages: code for manipulating the domain model and code for specific applications. The code for model manipulation is produced by the Eclipse code generation facility, including the generation of Java packages, classes, methods, attributes, and references. The Eclipse code generation facility is extended to generate code specific to the domain application, create deployment configuration, and integrate with components and web services specified in the application model.

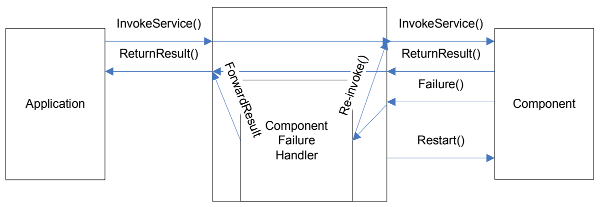

Fig.3. Generated failure handler for components

The dependability specification dictates how component failures are handled in the generated code. Figure 3 illustrates the scheme of handling component failures. A "Dependable Delegate" is generated to handle all interactions with a component. The "Dependable Delegate" defines all the interfaces declared in the component. An invocation (such as "invokeService()") is passed to the delegate which in turn routes it to the component to invoke the actual service. Invocation results are returned to the invoking application via the delegate. For the application, the function of the inserted delegate is transparent and the overhead is minimal since an invocation simply passes through the delegate if the invocation is successful. If the invocation fails, the "Dependable Delegate" will act upon the failure based on the failure handling strategies declared in the dependability specification. Actions may include re-invoking the service, restarting the component, or simply passing the failure along. The handling of web service failures follows similar scheme.

As an example to illustrate the generation of failure handling code, the invocation of logging a call detail record without failure handling, *billingComp.log(aCDR)*, is transformed to the following code template to handle component failure:

```
if (depSpec.isValueErrType()) {
      //call billing service to log call-detail-record (aCDR)
       result = billingCompProtector.log(aCDR);
       if (result == depSpec.getErrValue() ) {
                   // handle fault based on strategies
      }
         return result;
}
if (depSpec.isExceptionErrType()) {
       try {
              //calling the billing service
              return billingCompProtector.log(aCDR);
       }
       catch (...) {
             //handle specific fault based on strategies
        }
        catch (Exception anException ) {
            // handle general fault based on strategies
        }
    }
```

Different types of failure handling strategies in the DependabilitySpec are used to handle different types of components and web services based on the strategy specification. Web services, for instance, may require the handling of connection time out. If the same web service is deployed on two hosts, one of the handling strategies is to pick one host to invoke the service first and try the other one if the invocation fails. The failure handling strategies are automatically translated into implementation code and integrated with the application code.

We choose J2EE as the target platform for deploying domain applications. To generate and package code for an application, the code generation facility first creates a J2EE enterprise application project and populates the project structure with skeleton code. The code generation facility then interprets the application model and its model elements to produce corresponding Java code and J2EE deployment descriptors. The properties of receiving SMS and MMS messages in the modeled message component, for instance, are interpreted by the code generation facility to produce code to invoke the common utility services designed for receiving SMS and MMS messages.

The generated code also provides interfaces and extension points for the integration of customization code to further extend the functionality of the generated application. The generated application code, customization code added through the extension points defined in the generated code, and runtime libraries for service components are packaged as standard J2EE applications to be deployed on application servers. A browser-based test client is also generated to facilitate the validation of the generated application.

3.3. An environment for modeling domain applications

We developed a set of tools for modeling domain applications and generating application code. Although the formal specifications of domain meta-models provide complete information for constructing domain applications, it is inconvenient for application developers to model applications and services directly. A model-driven approach is taken to build such a model-based development environment.

Models are created for graphical definitions and tool definitions. These models are combined with the domain meta-models to serve as the foundation to build the GUI-based development environment. A graphical definition model defines the visual aspects of domain models in the development environment. It defines the graphical notations (icons, nodes, and connections) commonly used for visual representation of domain elements in the mobility service domain. A tool definition model is also used to specify tool elements (e.g. palette elements) for nodes and linkages for the model editor.

Mappings are defined to bind the elements in the domain model, graphical definition model, and the tool model to constrain and guide the creation and composition of model elements to build application models. While the graphical definition and tooling models provide visual aids to build application models, the semantics for model construction is contained in the domain meta-models. Mapping among the models is to ensure the correctness of the application models created in the development environment.

We extended the Eclipse Modeling Framework (EMF) [7] and Graphical Modeling Framework (GMF) [8] to implement the model-driven development environment for mobile services. The design of the development environment is entirely driven by the domain meta-models. While the development environment enables application developers to construct application models using domain notations, the application models are validated by using the domain meta-models. For instance, the composition of two components must have matching interface and the implementation artifact must be supplied before application code can be generated.

The mobile service creation framework is implemented as a set of plug-ins in Eclipse, an extensible open source development platform and application frameworks. The domain modeling of mobile services is based on the Eclipse Modeling Framework and the design of the development environment is based on the Eclipse Graphical Modeling Framework). The Eclipse Web Tools Platform (WTP) is used for generating J2EE projects and packaging domain applications for deployment. The packaged mobile applications and services are deployed on standard-based J2EE Application Servers.

4. Building and deploying message-based mobile services

The model-based component framework was employed to develop and pilot a number of mobile applications and services for several wireless service providers. We collaborated with these wireless service providers to provision the wireless networks to route SMS and MMS messages and support the deployment of mobile applications that are designed and generated from our model-based component framework. Results from the model specification, generation, and deployment of mobile applications and services showed substantial improvement in software development productivity over our own code-centric approach.

Figure 4 illustrates a message-based mobile application that checks flight status and provides weather information for mobile users. A mobile station sends an SMS message with flight numbers and carrier names to the service provider (predefined SMS code). The application extracts the content from the message, invokes the flight status web service, composes an MMS message containing flight status with advertisement, and sends the composed MSM message back to the sender. The usage of the service is also logged.

An instance of WSDepSpec (dependability specification from web services) is created and associated with FightStatusService web service. Redundant web services (multiple web service hosts offering the same service) are specified using this instance and they are configured to be invoked in a round-robin fashion. Similarly an instance of ComponentDepSpec is associated with logging services to ensure usages are recorded reliably.

5. Related work

The Generic Modeling Environment [9] provides a configurable toolkit for specifying meta-models and creating domain-specific modeling environments from meta-models. The modeling paradigm defines the family of models that can be created using the generated modeling environment. Cadena [10] describes an integrated environment for building and modeling CORBA Component Model (CCM) in software systems. It provides facilities for defining component types using CCM IDL, assembling systems from CCMcomponents, and verifying correctness properties of models of CCM systems derived from CCM IDL. The author in [11] describes a modeling infrastructure for integrating different modeling techniques and the transformation of models based on meta-models. Its code generation produces implementation skeletons and platform glue code for developers to integrate with business logic. A domain-specific modeling language is described in [12] to specify component interface definitions, component interactions, and component deployment. Authors in

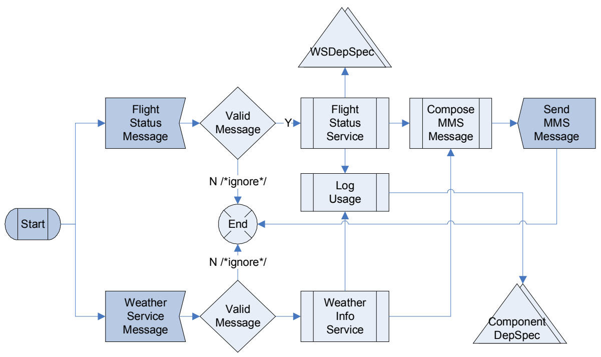

Fig.4. A message-based mobile application model

[13] describes model-typed interfaces based on generic interface parameters to facilitate the transfer of complex structured data between components and compatibility checks of model-typed interfaces at assembly time.

Authors in [14] describe a source-to-source compiler technique that applies source code transformation rules to introduce code modification for fault detection. Data and code duplication was introduced to improve software dependability. Authors in [15] introduce a multiple view modeling approach to ensure component dependability. Component models are specified with four modeling perspectives: component interface, static behavior, dynamic behavior, and interactions. Consistencies and dependencies among the models are established and maintained to achieve system dependability. In [16], self healing mechanism is proposed to build reliable systems based on connectors. Besides synchronizing message communication between tasks in a component, connectors are extended to support self healing by detecting anomalies in anomalous objects, reconfiguring objects in components, and repairing anomalous objects detected.

The framework described in this paper takes the integrated modeling of application domains, components, web services, and their dependability. The framework also automates the generation and integration of both application code and a failure protection code from the domain models. Compared to previous approaches, the integrated framework described in this paper allows the specification of component and service dependability at the model level independent of languages and platforms. Platform and language specific code is generated from models to implement both the functional and dependability specifications.

6. Conclusion

This paper describes a model-based approach for the specification and generation of dependable mobile applications and services. The key contributions of this work are the integrated specification of functions and dependability in domain-specific models, the automatic generation of protection mechanisms to minimize component and web service failures, and the generation of applications from domain models to reduce coding efforts. The pilots of this framework with applications developers and service providers showed productivity improvement in developing and deploying mobile services. Legacy systems and wireless infrastructure services are abstracted as model-level components in a domain-specific modeling environment to automate integration and validation. While most of the modeling technologies generate skeleton code, this integrated framework captures domain models and domain rules to further automate the generation of applications. Dependability specification is an integrated part of the domain meta-models to generate domain applications with improved availability and reliability. Compared to our previous code-centric approach to developing mobile applications and services, substantial improvement on productivity and dependability was achieved with the integrated framework and its development environment.

Integrating components and web services into model-driven development can be a very effective solution to improve both software productivity and quality. With the specification of component and service dependability, further improvement on software quality and reliability can be achieved. Besides the application domain discussed in this paper, we believe the approach can be effectively applied to other domains for application development. One of the goals for our future work is to extend this framework for the specification of performance and real time requirements.

7. References

[1] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 2002.

[2] World Wide Web Consortium (W3C), *Web Services*, http://www.w3.org/.

[3] J. Greenfield et al., *Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools*, John Wiley & Sons, 2004.

[4] Object Management Group, *CORBA Component Model*, http://www.omg.org/technology/documents/formal/components.htm.

[5] T. Genßler and C. Zeidler, Rule-driven Component Composition for Embedded Systems, *4th ICSE Workshop on Component-Based Software Engineering: Component Certification and System Prediction*, May 2001.

[6] J. Dong, Model checking the composition of hypermedia design components, *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, 2000.

[7] Eclipse Foundation, *Eclipse Modeling Framework* (EMF) http://www.eclipse.org/emf/.

[8] Eclipse Foundation, *Eclipse Graphical Modeling Framework (GMF)* http://www.eclipse.org/gmf/.

[9] Institute for Software Integrated Systems (ISIS), *The Generic Modeling Environment (GME)*, http://www.isis.vanderbilt.edu/Projects/gme.

[10] J. Hatcliff, et al, Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems, *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*.

[11] O. Kath, An Open Modeling Infrastructure integrating EDOC and CCM, *Proceedings of the Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC'03)*.

[12] K. Balasubramanian, J. Balasubramanian, J. Parsons, A. Gokhale, D. C. Schmidt, A Platform-Independent Component Modeling Language for Distributed Real-time and Embedded Systems, *Journal of Computer and System Sciences, volume 73, issue 2*, 2007.

[13] G. Schmoelzer, E. Teiniker, C. Kreiner, M. Thonhauser, Model-typed Component Interfaces, *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06)*.

[14] M. Rebaudengo, M.S. Reorda, M. Violante, M. Torchiano, A Source-to-Source Compiler for Generating Dependable Software, *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, 2001.

[15] R. Roshandel and N. Medvidovic, Multi-View Software Component Modeling for Dependability, in Architecting Dependable Systems II, *Lecture Notes in Computer Science*, 2004.

[16] M. E. Shin and D. Cooke, Connector-Based Self-Healing Mechanism for Components of a Reliable System, Workshop on Design and Evolution of Autonomic Application Software, ICSE, 2005.