Transformations of a SIP Service Model

Bruggeman, W.

Award date:
2008

TU/e, Eindhoven University of Technology

Mathematics and Computer Science department

Software Engineering and Technology group

master thesis

Transformations of a SIP Service Model

by W. Bruggeman

graduation tutor: dr.ir. T. Verhoeff, TU/e
graduation supervisor: prof.dr. M.G.J. van den Brand, TU/e
supervisor: ir. M. Huijsmans, Ericsson Telecommunicatie BV

Eindhoven, June 2008

Abstract

Does capturing more of the semantics of an IMS Service in the early phases in a model and using this model in later phases of the service lifecycle result in more flexibility and faster TTM without loss of quality?

This master thesis analyzes the above question through an investigation of an IMS Metamodel defined by the Ericsson Service Composition Environment, an environment for composition of IMS Services. In addition, transformations from IMS Services modeled as UML State Charts into both SCE implementations and SIP Applications are investigated. This includes an analysis of the transformation design processes as well as techniques to add SIP Semantic information to a State Chart metamodel.

While a transformation from a SIP Model into a SIP Application was successfully performed, it was found that the transformation as such should be seen as part of a larger design process to be able to answer the question. These gaps have been identified and topics for further research are included.

Acknowledgments

The author would like to thank the following persons: Martien Huijsmans for being my tutor at Ericsson. Tom Verhoeff and Mark van den Brand for their support as graduation tutor and graduation supervisor at the Eindhoven University of Technology. The SCE design team at ETM for their amazement when asked yet another silly question on SCE. All the colleagues at the System Management and CA MMS departments. Arie Catsman, and all other proof readers.

Contents

1 Introduction

This document is the result of the master thesis project carried out at the System Management department at Ericsson Telecommunicatie BV, the Netherlands.

This master thesis is also the last part of the Master of Science degree at Eindhoven University of Technology.

Within Ericsson there is an interest in Model Driver Engineering. These developments at modeling level are seen as a means to improve maintainability and flexibility.

As part of a research project at Ericsson Research Germany, the Service Composition Environment was developed. This modeling environment uses an IMS Service Metamodel

The System Management combined both facts and provided a master thesis opportunity which ultimately resulted in this Master Thesis document.

This document can be split in two parts:

- The first part describes the assignment (chapter 2), provides an introduction on IMS (chapter 3), an introduction on SCE (chapter 4) and finally describes an IMS Service (chapter 5).

- The second part describes the investigation on the SCE State Machine Transformation (chapters 6 and 7), the investigation on the Repleo State Machine (chapter 8 and 9), an investigation on SIP Semantics (chapter 10), and finally an overview of the design processes used for the transformations (chapter 11).

The conclusions can be found at the end of chapters 7, 9 and 10. Chapter 12 contains the overall conclusion and includes recommendations for further research.

2 Assignment

2.1 Introduction

The objective of this chapter is to provide a background and to describe in more detail the context and scope of the graduation project.

The graduation project is based on the *initial assignment* described in the “Student Assignment” [1], and the *engineering assignment* described in the "Detailed Student Assignment" [2].

The initial assignment of System Management describes an investigation into an IMS Service Model defined by the Ericsson Service Composition Environment and includes sub investigations into component frameworks and the service life-cycle.

Approximate halfway the project it was observed by the graduation tutor and graduation supervisor that the engineering aspects of the initial assignment would be difficult to measure. Together with System Management we introduced the engineering assignment to address these concerns.

The engineering assignment requests an investigation of automatic transformations of an IMS Service Model into a Service Composition Template implementation. There were no difficulties shifting the investigation.

Based on feedback from the graduation supervisor, the engineering assignment was extended with an investigation of automatic transformations of an IMS Service Model into a Java SIP Application based on the syntax safe generation system Repleo.

The main focus of this thesis is the engineering assignment. As a result some of the research questions of the initial assignment were removed from the scope of this work.

2.2 System Management

The graduation project was performed for Ericsson Telecommunication B.V. at Rijen, The Netherlands. The stake holder is the PDU VAS System Management department.

Ericsson is a world-leading provider of telecommunications equipment and related services to mobile and fixed network operators globally.

The (simplified) operational organization is based on business units and market units. Business Units create offerings and Market Units sell offerings. A Business Unit consists of Development Units and a Development Unit consists of Product Development Units.

- The Business Unit Multimedia (BMUM) has the overall responsibility of the multimedia solutions portfolio.
- The Development Unit Multimedia Products (DMMP) provides multimedia products.

- The Product Development Unit Value Added Services (PDU VAS), located in Rijen, The Netherlands, provides products within the scope of value added services.

The System Management at the PDU VAS is responsible for the system definition, general design and documentation for development of products.

As the System Management at the PDU VAS is part of a larger System Management organization, whenever System Management is mentioned in this document, this should be seen in the context of the PDU VAS.

2.3 Background

This chapter provides a background for the assignment. The points mentioned address the "why" and "why now" of the assignment.

2.3.1 White Box

System Management sees a tendency towards white box systems.

A Black Box is a system of which only the interfaces and behavior are known. The content of the box is not known and cannot be changed. The Black Box has a known functionality which cannot be changed.

The White Box is a system from which the interfaces and behavior are known, as well as the components inside the box. The components inside the box could be white or black boxes. The components inside the box can be replaced or reordered, which results in a different behavior. A White Box allows flexibility at the cost of control as a specific behavior can no longer be guaranteed.

A telecom service can be represented as a black box system. A telecom service has a known functionality and cannot be changed. By modeling a telecom service as a white box flexibility is added. Components inside the box become known and can be replaced or reordered. This allows matching specific requirements of a customer and allows adapting to changing requirements over time.

2.3.2 IP Multimedia Subsystem (IMS)

Currently there is an evolution of the telecommunication network in which the core network is changing from switched based telecommunication networks to IP based telecommunication networks. These IP based telecommunication networks are referred to as Next Generation Network, or, more specific, as the IP Multimedia Subsystem (IMS) telecommunication network.

The purpose of IMS is to provide a framework in which multimedia sessions can be established and value added services can be deployed.

Ericsson has an IMS product that includes core network layer components, as well as service layer components. A service development solution is available for creation of telecommunication services.

While for the current generation of telecommunication networks the design processes and frameworks are mature and fixed, for IMS these processes and frameworks are still being defined and new design methodologies and architecture frameworks are being introduced.

The assignment should be seen in the context of IMS. To reflect this, the term “IMS Service” is used, instead of, for example, “telecommunication service”.

2.3.3 Product Life Cycle Management

Within Ericsson the Product Life Cycle Management (PLCM) describes the process to create a product or solution. This includes a life cycle process ranging from the definition of a market opportunity to the phasing out of a product. Part of the PLCM is the discipline Network System Modeling (NSM). The objective of NSM is to specify the architecture of a solution. The output from Network System Modeling will be input for development projects.

Currently, the output from the Network System Modeling specifies a system (service), but the (detailed) modeling of the service design and implementation is not performed until the next phase in the PLCM process (the Design & Test Product phase).

System Management believes that modeling in an earlier phase (that is, during NSM) may improve management of complexity and add to the flexibility of changes later in the life cycle.

As an example in a broader scope: The PDU provides standard products and solutions. The Market Unit takes a selection of these standard products and solutions, adds customization to comply with specific customer requirements and delivers a complete solution to the customer. Being able to customize a product with minimal redesign improves the flexibility. Having influence on the modeling improves the manageability and supports the flexibility of the product.

2.3.4 Model Driven Engineering

System Management sees developments at the modeling level. For example Model Driven Architecture and Model Driven Design. Capturing (more) information in a model and using this model as a means for the system and software architect is seen as an enabling element towards the Market Unit.

An interest in Model Driven Engineering can also be seen at other levels. For example, DMMP sees developments at the modeling level as a means to improve the manageability of the product portfolio and underlying system components. A research project is started in this area, but no results are available yet.

2.4 Context / Scope

2.4.1 IMS

The scope of the assignment is based on the Service Composition Environment (SCE) prototype. SCE is discussed in paragraph 2.4.2. The relevant technologies and implementations for SCE are indicated below.

- The scope of the service is a session oriented telecommunication service in the IMS architecture. (Also referred to as an IMS Service.) The IMS framework [3] defines a functional component called SIP Application server on which IMS Services can be deployed and executed. SIP is the signaling protocol used in IMS. Note that the IMS specification does not describe the implementation or technology choices within the SIP-AS.

- The scope of the implementation and technology for the SIP Application Server is related to the SCE prototype. This defines a Java EE application server and the JSR-116 or JSR-289 Sip Container. The logical components are shown in Figure 1. Although not part of the IMS architecture, the figure also shows the Composition Design Environment that is used to compose IMS Services.

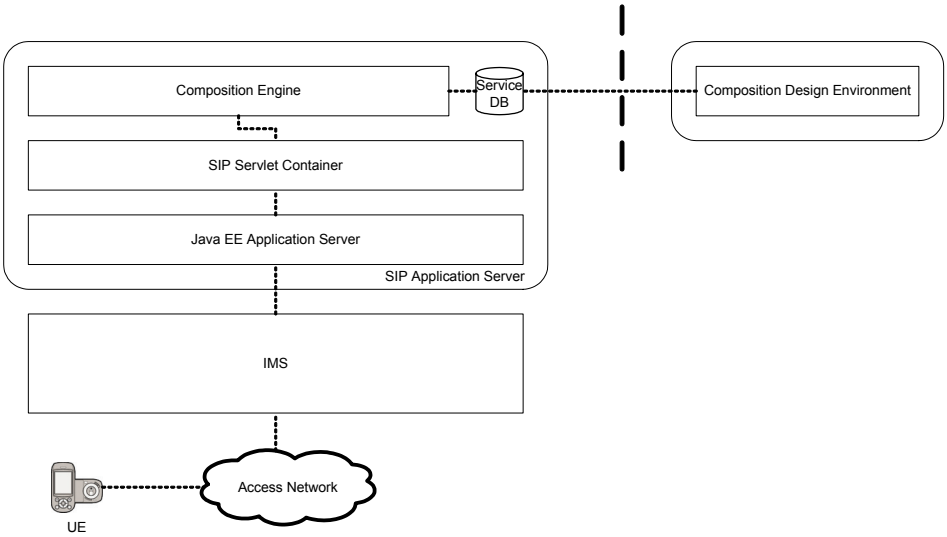


Figure 1, SIP-AS Logical Components

- The SIP Container will be JSR-116 [12] or JSR-289 [13] (pending availability of a finalized specification and implementation). In particular the Sailfin [15] implementation of the SIP Container. Note that the JSR-289 specification is not yet finalized nor is an implementation available. The Sailfin implementation is also not yet finalized. The current implementation of the SCE prototype uses the JSR-116 container.

- The Service Composition Environment (SCE) is defined by the Multi Service Architecture documentation (see chapter 2.4.2) and the SCE prototype, documentation and sources.

2.4.2 MSA

As part of the Multi Service Architecture (MSA) research and in co-operation with the University of Bremen, Ericsson Research Germany (EDD department in Aachen) has developed a Service Composition Engine (SCE) environment.

The SCE allows composition and execution of IMS Services.

Composition of IMS Services is based on:

- Constituent Services. These are atomic service components that represent a single task within a service. Multiple Constituent Services can be grouped together (in a skeleton) and accessed as a Constituent Service.

- Composition Templates. These represent complete IMS Services or specific tasks and consist of Constituent Services and Composition Templates.

IMS Service Execution is based on:

- The service flow specified in a Composition Template.

- Based on the referenced Constituent Services a service implementation is selected and used to execute a Composition Template step.

The SCE components of the current prototype are listed below:

- SCE Composition Engine
- Service Database
- SCE Composition Design Environment

These SCE components are shown in Figure 2.

The SCE Composition Engine is a J2EE / SIP Container based Execution Engine to execute IMS Services.

The SCE Composition Design Environment is an IMS Service development environment. The created IMS Services are stored in the Service Database.

As a result of the MSA research, prototypes are available of the Composition Engine, Composition Design Environment and Service Database.

The SCE includes a service model that is input for the investigation.

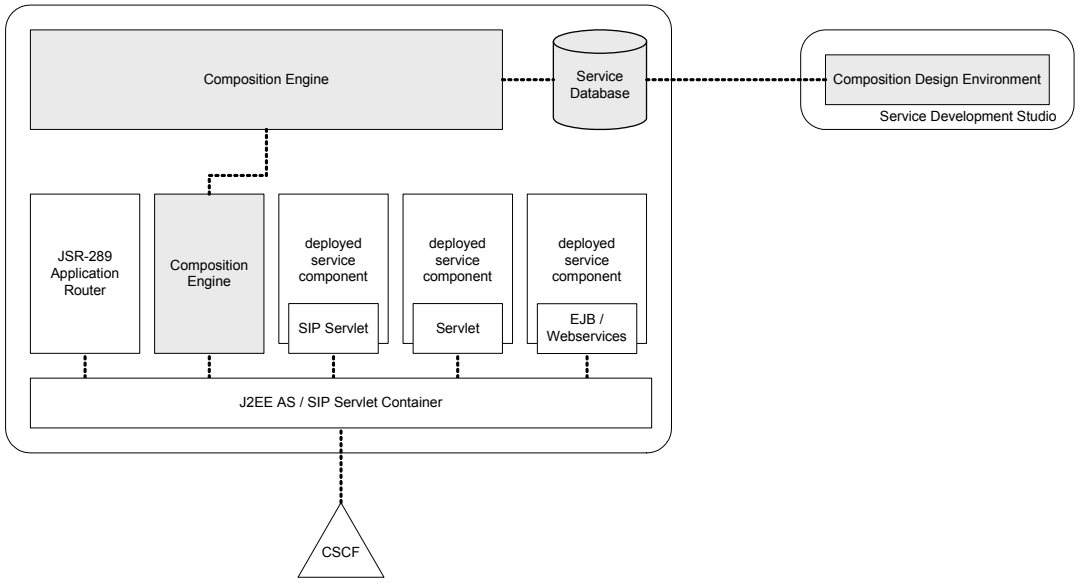


Figure 2, Service Composition Environment

The Constituent Services as used by the SCE provide a description (a service description and constraints) and are neither executable nor deployed components.

The deployed service components (for example deployed SIP Applications or Web Services) are identified through a set of properties.

The binding between constituent services and the deployed service components is based on these constraints and properties. A set of constraints that describe the constituent service is specified during service composition. During execution, the Composition Engine finds a deployed service component whose properties match with the constraints and invokes this deployed service component.

The branding of an IMS Service is inflexible as it is typically done through configuration and not during service composition. Using the constraint mechanism of SCE, choosing specific branded deployed service components can be done during service composition.

An example would be to use a property named "provider". By changing the value of this property during the composition it is trivial to switch between different sets of branded service implementations.

2.4.3 Services

The IMS Services used in this project are based on the TISPAN Multimedia Telephony Services specification [3]. Simplification and/or implementation of these IMS Services are performed as part of the project.

2.4.4 Ericsson Tools and Processes

Within Ericsson there is a strong focus on the Eclipse IDE. Further, UML based tools are used.

In this project we use the Ericsson SCE prototype and the Ericsson Service Development Studio 4.0 [18].

2.5 Problem Description

Does capturing more of the semantics of an IMS Service in the early phases in a model and using this model in later phases of the service lifecycle result in more flexibility and faster TTM without loss of quality?

The SCE prototype provides a framework to compose and execute services allowing composition in an earlier phase. What kind of service model does SCE use and how does SCE compare to alternative frameworks?

In the current implementation of services, state machines play an important role. Can support for state machines be provided in SCE?

2.6 Initial Assignment

This chapter provides a more detailed description of the initial assignment as described by the "Student Assignment" [1].

The assignment questions listed below are further discussed in the following paragraphs.

1 Investigate the IMS Service Model defined by SCE.

2 Investigate the pro's and con's of component frameworks.

3 Describe the life-cycle of a service.

As an indication, question 1 is expected to take 70%, question 2 is expected to take 20% and question 3 is expected to take 10% of the assignment effort, within the scope of these three questions.

2.6.1 Service Model

Investigate the IMS Service Model defined by SCE.

- Analyze and extend the IMS Service Model defined by SCE.
- How does the IMS Service Model defined by SCE compare with alternative service models?

2.6.1.1 Initial Assessment

SCE provides a framework and a prototype of a Composition Design Environment and Composition Engine. SCE contains a service model that allows modeling, storage and deployment of IMS Services. The service model is based on Composition Templates (service flow descriptions), Constituent Services (service elements) and control (flow) elements.

A prototype of SCE is available. This includes source code and some documentation. The available documentation appears to be limited and the Java source code appears to contain only limited comments. The designers involved with the creation of SCE (and thus the service model) may be available to answer questions.

The purpose of the service model is to represent IMS Services. The suitability of the IMS Service Model defined in SCE to represent IMS Services can be measured in two ways:

- By identifying the properties of an IMS Service and verifying if these properties can be represented in an IMS Service Model.
- By composing existing IMS Services (use cases) using the IMS Service Model.

IMS Services have certain properties which can be found by studying the IMS specifications and the J2EE / JSR-289 framework. Such properties are for example the a-synchronous behavior, the forking of sessions and the forwarding of SIP requests. Next, it can be investigated if these properties can be represented by the service model.

A set of IMS Service specifications/descriptions is available. These are taken from the TISPAN Multimedia Telephony standard [3] and use cases from System Management such as the Managed Telephony IMS Service. Using these use cases it can be investigated if these services can be composed using the SCE service model.

Composing IMS Services using the service model is done by analyzing use cases, identifying the constituent services and modeling these components in skeletons. This should not only result in identifying any discrepancies between the required components and what is allowed in the service model, but should also result in a set of common constituent services and an understanding of best practices for service composition.

If shortcomings are found in the SCE service model, these should be documented and a solution to the service model investigated. Depending on the expected effort the implementation of the SCE service model should be updated in the prototype.

The question "how does the service model compare with alternative models" may be difficult to answer. Preliminary investigation indicates there are several service composition models. Among others: WS-BPEL, WS-SDL, XPDL or BPML. These models are typically based on web services. While IMS Services are similar to web services there are differences due to the IMS domain. To compare the SCE model with alternative models, a (small) selection of alternative models should be decided on. Further, these alternative models may need to be extended to allow IMS properties to be represented. Finally, the question arises on what properties the SCE model should be compared.

2.6.1.2 Approach

- Study the SCE prototype (source code and documentation) to extract the IMS Service Model. Preliminary investigation indicates the documentation is limited. The implemented service model may be based on an existing standard. As this investigation is based on an implementation and not on a specification it may be difficult to extract a (complete) service model as the implementation is probably a subset of a specific service model. The SCE designers may provide input on the SCE service model.

- Study the IMS specification, as well as the SIP protocol and JSR-116/JSR-289 documentation to extract IMS Service specific properties. Preliminary investigation suggests these properties are mostly related to the SIP protocol and JSR-116/JSR-289 containers. Examples of such properties are the a-synchronous nature of the session, the ability to fork a session and the ability to include SIP Servlets, HTTP Servlets and EJB components in a service.

- Analyze if these properties can be represented using the SCE service model.

- Study the TISPAN Multimedia Telephony specification and decide on which services to use. Make a list of typical service scenarios such as identification, diversion, waiting, barring and conference calls. The selection of services should represent these scenarios. Additional use cases available from System Management projects can be added.

- Compose the use cases using the SCE prototype. This involves an analysis of the use cases to identify common components to represent as a constituent service. Some functionality could be identified to be implemented outside of the SCE service model scope. For example, certain functionality of a service it could be handled in the CSCF node or through the JSR-289 Application Router in a different (non SCE) SIP Application.

- The actual composing of the use cases is done using the SCE prototype. Besides a proof of concept showing that IMS Services can be composed using the SCE service model this task also provides a basic set of constituent services and skeletons as well as an insight in composing IMS Services using the SCE prototype.

- Limitations in the SCE service model may be found. It should be investigated how to remove these limitations and, if feasible, the service model description and prototype should be updated to reflect these changes.

2.6.1.3 Expected Result

- A description (specification) of the service model used in SCE.
- The result of the analysis whether IMS Services can be modeled using the SCE service model.
- A set of constituent services, skeletons and an overview of the experiences and best practices for service composition.
- If limitations are found in the SCE service model, an updated service model and, if feasible, an updated SCE prototype.

2.6.1.4 Risks

Based on the preliminary research the service model used is not documented and may need to be extracted from SCE documentation and the SCE prototype. This may prove difficult.

If limitations are found in the SCE service model, and the SCE prototype needs to be updated, this may be time consuming.

2.6.2 Component Frameworks

Investigate the SCE component framework.

- How does the SCE component framework compare with alternative component frameworks?
- Are there gaps in the (specification of the) SCE component framework?

It is expected the assignment will focus on a single bulleted question.

2.6.2.1 Initial Assessment

Preliminary investigation indicates there are gaps in the process from development to deployment of Constituent Services. For example, the SCE framework and prototype do not discuss how the Constituent Services should be implemented, nor how these should be deployed or how the descriptions and properties are stored in the database. Further, the SCE prototype is currently based on JSR-116. In JSR-289 an Application Router component is specified. It is unclear how the Application Router component and the Composition Engine should interact.

During the investigation of the IMS Service Model more limitations may be identified.

Depending on the result of the investigation of the IMS Service Model (see paragraph 2.6.1) it will be decided to focus on a comparison with alternative framework or to investigate the limitations of the SCE component framework.

Next to the investigation of the SCE Service Model, this is a minor investigation.

Preliminary investigation indicates two possible directions for alternative frameworks.

The first direction could be based on the BPEL work flow language. This could involve three steps. In the first step an IMS Service modeled in UML is translated to BPEL. In the second step the IMS Service represented in BPEL can be composed (changed). In the third step the IMS Service can be deployed using BPEL execution engines. This would provide an alternative to the SCE framework.

The second direction could be based on the idea of composing using Composition Templates (that could be represented using SIP Applications) and Constituent Services (a certain binding of the Service Elements) using an execution engine. This SCE framework could be compared with a Model-View-Controller framework such as the Spring framework.

Preliminary investigation of the SCE framework (and processes) indicates a gap concerning component development and service deployment.

Not only is a technical point of view important. Also a process point of view should be taken into account. This might also include packaging and version management.

2.6.2.2 Approach

First, the existing SCE framework should be investigated. What does the component architecture look like? How are components developed? How are Composition Templates composed? And how are services deployed? This investigation is based on the SCE documentation and SCE prototype.

Depending on the direction the investigation will have a different focus. Possible areas of investigation are:

Investigate what the requirements are for deployment. Not only technical aspects should be taken into account, but also process and life-cycle aspects are relevant. "Who" performs "what" task, and how does this impact the deployment process. This overlaps with the question referenced to in paragraph 2.6.3.

Define the properties that a component framework should support. These properties are expected to be mainly based on the IMS Service Model. How do the constituent components, Composition Templates and their bindings impact a component framework? Does a component framework impact the service model? On what level should the interface layer be defined? (E.g. is this on JSR-289 level or on SIP-AS level?)

Identify possible alternative component frameworks and decide which one to investigate further

Investigate the deployment using the alternate component framework.

2.6.2.3 Expected Result

The expected result depends on the direction of the investigation.

- A description of the SCE component framework.
- The result of the analysis of an alternative component framework.
- The result of the analysis of a limitation in the SCE framework.

2.6.3 Service Life-Cycle

Describe the life-cycle of a service.

- What process phases are there?
- What stakeholders are there?
- What impact has the SCE framework on service design?

2.6.3.1 Initial Assessment

In the life-cycle of an IMS Service different process phases and stakeholders can be identified. An example of the life cycle and stakeholders of an IMS Service is given in Table 1.

Life-cycle step	Stakeholder	Purpose
component design	PDU architect	Specifying the (interface) of service components
service composition	PDU architect	Composing a service from service components
component implementation	PDU designer	Implementing a service component. This view could represent code generation.
service integrator	MU integrator	Deploying the services and service components on an execution environment. This may be a view of an assembled service, as well as a view represented by deployment configuration file generation.
customer service composition	Customer architect	The customer may want to perform service composition. This view may be a limited or restricted view compared to the architect view.

Table 1, Example of life cycle and actors

The overview of the life-cycle and stakeholders could be further extended.

From a life-cycle step and stakeholder point of view, how is the relation and interaction with the SCE framework.

What views and tools are needed to fulfill the stakeholders and process needs? Do these views impact the SCE framework and service model?

Next to the investigation of the SCE service model, this is a minor investigation.

2.6.3.2 Approach

- Investigate and describe the life-cycle steps and stakeholders. Correct or extend the list given in Table 1. This will probably involve study of the current service design process documentation and interviews of System Management designers and experts.
- Correlate the identified life-cycle steps and stakeholders with the SCE framework and service model. Identify what "views" are needed.

- Investigate whether the views impact the SCE framework or service model. Visa versa, how does the SCE framework impact the service design process?
- What kind of techniques and tools are available to generate the views?

2.6.3.3 Expected Result

- A description of the life-cycle and stakeholders of an (SCE) IMS Service.
- A description of the views needed by the various stakeholders.
- An overview of the impact between the SCE framework and the views.

2.6.4 Result

- The Master Thesis

  Describing the results of the investigation.

- A Prototype

  A set of Composition Templates that represent the composed set of selected use cases.

  This includes a set of implemented and deployed Constituent Services, although no full functionality is required.

  An updated SCE prototype with an enhanced service model.

2.7 Engineering Assignment

This chapter provides a more detailed description of the engineering assignment as described in "Detailed Student Assignment" [2].

2.7.1 Introduction

There is an existing Service Composition Environment (SCE) available. This has been studied and a model has been created.

There is a service consisting of Personal Greeting Service (PGS) and Call Forwarding On Busy (CFB). These are implemented as regular SIP Servlet (Java) and in C++. Descriptions exist of both services, but for the study a simplified version will be used.

2.7.2 Study

Design and implement the above service using the existing Service Composition Environment (SCE).

- Define the components (called Constituent Service in the current Model). What is the interface that these components expose to the Service Composition Actor?

- One or more of the components will be a SIP Application that consists of B2BUA and Forking SIP Servlets. Does this impact the service composition or is it hidden for the Service Composition Actor?
  If exposed and undesirable, in what way can SCE be enhanced.

- Investigate how parallelism and the event driven properties of the components can be supported in SCE.

- In the current implementation state machines play an important role. Investigate if and how support for state machines can be provided in SCE.

- Error handling plays an important role to provide robust services. Investigate if the current error handling capabilities meet the needs of the identified components.

Investigation includes: problem definition, analysis, design and implementation.

2.7.3 Model Transformation

The engineering assignment settled on the investigation whether a UML Model representing a telecommunication service can be automatically transformed into an implementation of said service.

This investigation was performed for two distinct areas.

1 A transformation from a UML State Chart Model into a State Machine implementation for the Service Composition Environment.

2 A transformation from a UML State Chart Model into a State Machine implementation for a Java JSR-116 based application server utilizing Repleo for source code generation.

An overview of the initial observations and approach for the SCE State Machine Transformation investigation and the Repleo State Machine Transformation investigation can be found in chapter 78 and chapter 8.

2.7.4 Result

- The Master Thesis

  Describing the results of the SCE State Machine Transformation and the Repleo State Machine Transformation.

- Transformation Artifacts

  Including:

  - Reference Implementations (SCE Templates, Java Source Code)
  - All input artifacts (Stored UML Models, XMI Documents)
  - Transformation Specifications (XSLT Documents, Repleo Templates)
  - Generated Source Code
  - Auxiliary Documents (XML Schema Documents, Testing scripts)

3 IMS

3.1 Introduction

This chapter provides an introduction to IMS with a strong focus on IMS applications in which the SIP Application Server is part of the IMS session.

3.2 IP Multimedia Subsystem (IMS)

IP Multimedia Subsystem (IMS) is a framework standardized by the 3rd Generation Partnership Project (3GPP) in collaboration with the Internet Engineering Taskforce (IETF). The purpose of IMS is to provide an open architecture and platforms for multimedia services. [10]

IMS provides support for:

- IMS Sessions and IMS Services
- Roaming
- Quality of Service
- Access (Network) Independence
- Operator Policy Control of IMS (Services)

IMS provides a framework that allows rapid service creation and deployment. IMS Services itself are not standardized by 3GPP to allow compatibility with commercially available IMS Services. IMS Services can be deployed in a vendor independent manner.

Roaming allows subscribers to use IMS Services when roaming outside of the Home Network.

IMS allows negotiating of the Quality of Service by the user and operator during the session as well as during the session establishment.

IMS supports access independence. The IMS Services should be available regardless of the access of subscribers.

3.3 Architecture

The 3GPP standards are a specification of functions and interfaces between these functions. The functions and not actual nodes are specified. Telecommunication companies can combine (or split) functions between nodes.

3.3.1 3GPP Logical Architecture

The 3GPP logical architecture can be seen from a telecommunication perspective based on a split between an Access Network and a Core Network.

A Public Land Mobile Network (PLMN) is a telecommunication network deployed by an operator. Such a network can contain a circuit switched domain, a packet switched domain and an IMS domain.

3GPP logically divides a PLMN in an Access Network (AN) and a Core Network (CN) infrastructure.

The Access Network provides access for a subscriber to the core network and consists of physical entities that are in contact with the user equipment.

The Core Network provides support for telecommunication services such as user location management, network control switching and transmission mechanisms. The Core Network consists of a Circuit Switched, a Packet Switched and an IMS domain. See Figure 3. An overlap between domains is possible.

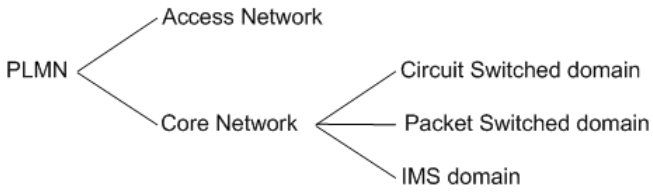


Figure 3, 3GPP Logical Architecture

3.3.2 IMS Logical Architecture

The IMS logical architecture can be divided in a Services Level, a Session Control Level and a Bearer Level and is represented in Figure 4. These levels are also referred to as Access Network, Core Network and Service Layer.

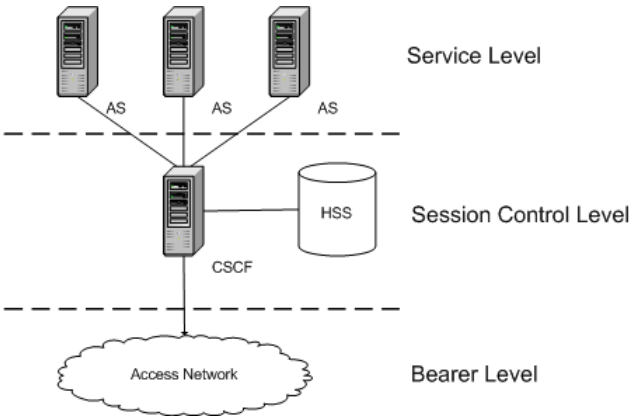


Figure 4, IMS Logical Architecture

3.3.3 IMS Functional Components

An overview of IMS functional components with a focus on services is shown in Figure 5. Functions such as the Breakout Gateway Control Function (BGCF) and Media Resource Function (MRF) are partly shown. Other functions such as the Interrogating Call/Session Control Function (I-CSCF) or the Subscriber Location Functions (SLF) are not shown.

The signaling plane and media plane are separated in the IMS architecture. The functional components shown in Figure 5 only carry signaling data.

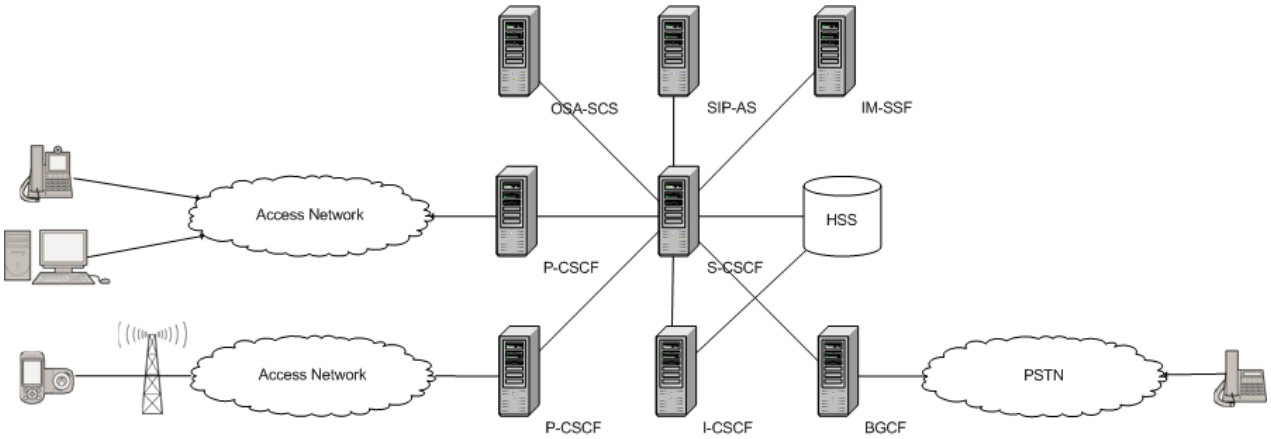


Figure 5, IMS Functional Components

The functional components are described below.

3.3.3.1 P-CSCF

The Proxy-Call/Session Control Function is the first contact point within IMS and behaves like a proxy. That is, it accepts requests and processes the requests or forwards the requests. The P-CSCF does not change the requests. The P-CSCF can forward requests towards the S-CSCF or the UE (user equipment). Typically, the request is forwarded to the S-CSCF of the subscriber.

3.3.3.2 S-CSCF

The Serving-Call/Session Control Function performs the session control for the UE (user equipment). A subscriber is linked to a specific S-CSCF. The functions of the S-CSCF are:

- accepts registration requests
- provides session control
- filter (intercept) requests and forward to services platform
- interact with the services platform (Application Server)
- receive and forward requests
- retrieve the service profile for a subscriber from the HSS

3.3.3.3 I-CSCF

The Interrogating-Call/Session Control Function is located on the border of an operators IMS network. The functions of the I-CSCF are:

- assign a S-CSCF to a user performing SIP registration
- route SIP requests towards the S-CSCF
- perform lookups in the HSS

3.3.3.4 BGCF

The Breakout Gateway Control Function allows interaction with a Public Switched Telephony Network (PSTN). The PSTN represents the "regular" fixed telecommunication network.

3.3.3.5 HSS

The Home Subscriber Server is a database that contains subscription related information for subscribers. The HSS contains data on:

- subscriber identification data (numbering and addressing information)
- subscriber security information (authentication and authorization for network access)
- subscriber profile information

3.3.3.6 Access Network

IMS is independent from the (IP Packet Switched) access network. It is possible to use the IMS Services from different connections such as LAN, GPRS or UMTS as well as different devices.

3.3.3.7 AS

The IMS architecture describes multiple application servers:

- OSA-SCS
- IM-SSF
- SIP-AS

All application servers are interfaced towards the S-CSCF using the SIP protocol.

The Open Service Access-Service Capability Server (OSA-SCS) provides an interface for the OSA / Parlay specification.

The IMS-Service Switching Function (IM-SSF) provides an interface for the Customized Applications for Mobile network Enhanced Logic (CAMEL) specification.

Both OSA / Parlay and CAMEL are existing telecommunication service specifications.

The Session Initiation Protocol-Application Server (SIP-AS) is an application server that allows to deploy IMS Services.

The OSA-SCS and IM-SSF provide an interface towards existing service platforms. It is expected new services are developed for the SIP-AS. The specification of the SIP-AS function and interfaces does not describe the actual implementation or technology of the SIP-AS.

3.4 Protocols

3.4.1 Signaling Layer

The Session Initiation Protocol (SIP) [11] is used to perform session control in the signaling plane. When used as part of the IMS framework, additional options and extensions are needed. For example, the IMS domain includes wireless access networks and presence features that put strict requirements on the session control protocol.

SIP allows setup, management and termination of multimedia communication sessions between devices. SIP is based on HTTP.

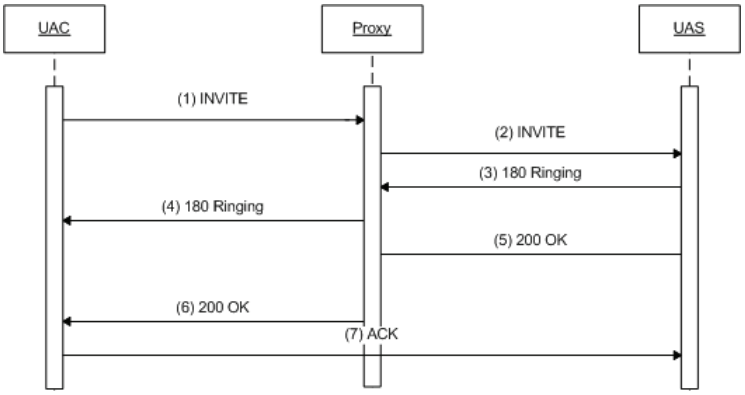


Figure 6, SIP Setup (Sequence Diagram)

The logical architecture of SIP is based on User Agents, Proxy Servers and Registrars. A User Agent Client (UAC) is a logical entity that can create and send a request. A User Agent Server (UAS) is a logical entity that can receive a request and generate a response. A User Agent can act both as a UAC and UAS. A Proxy Server is an intermediary entity that can both as a server and a client. The primary function of a Proxy is routing requests. A Registrar is a server that accepts and processes REGISTER Requests, and keeps track of the contact address of subscribers.

SIP allows user agent registering. During registration the public URI of the user is bind to a URI that represents the user equipment on which the user is logged on. Registration is mandatory for IMS. In addition of binding the public URI, registration also allows for authentication and authorization.

A SIP Dialog is the relationship between user agents and the messages send.

An example of a SIP Setup sequence in which the logical components can be seen is shown in Figure 6. This SIP Setup example can be compared to the example in Figure 8 which shows basically the same use case, but placed in the IMS framework.

3.4.2 Media Layer

The SIP protocol is used for session control. To describe and agree on the multimedia channel as well as to actually setup the multimedia channel other protocols are used.

- Service Description Protocol (SDP)
- Real-time Transport Protocol (RTP)

- Real-time Transport Streaming Protocol (RTSP)

The Service Description Protocol is used within SIP messages to describe the media session and is used, for example, during the session setup phase. The SIP and SDP protocols are part of the signaling plane.

The Real-time Transport Protocol and the Real-time Transport Streaming Protocol are used for transporting real-time data and for controlling the delivery of streaming media. The RTP and RTSP protocols are part of the media-plane.

3.4.3 The SIP Chain

During the session setup procedure a path between the originating and terminating user equipment is setup. This chain of (functional) nodes in a SIP Dialog or IMS Session is the SIP Chain.

3.4.4 User Identity

A subscriber has to register on the IMS network. During registration the authority and authentication of the subscriber is verified. The subscriber also binds his public user identity to a contact address (a SIP URI) that identifies the client on which the subscriber is logged on.

An IMS subscriber is internally identified by a Private User Identity. This Private User Identity is used for authentication, administration and accounting purposes. It is not used for routing. Multiple Public User Identities can be linked to a Private User Identity. The Private User Identity is not shown to the subscriber.

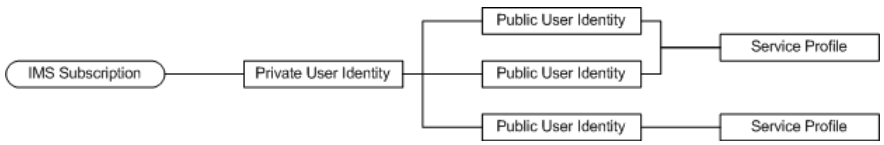


Figure 7, Relationship of User Identities

The Public User Identity (PUI) represents the subscriber from a routing perspective. This is the address that is used in address books or on business cards. The PUI in itself does not indicate where the subscriber can be reached. During registration the client on which the subscriber is logged on becomes bound to the PUI. The registration information if stored in the HSS. A PUI can be used to register multiple destinations. This allows session request to be forked to multiple clients. For example, an incoming voice session is routed both to a fixed client (phone) at home and to a mobile client. A Public User Identity can be compared to an e-mail address or a phone number (MSISDN) of a subscriber.

The relationship between User Identities is shown in Figure 7.

The Public User Identity is represented in the SIP URI or TEL-URI format. The TEL-URI format allows addressing using a phone number (MSISDN). Examples of Public User Identities are:

- `sip:firstname.lastname@operator.com`
- `tel:+31-6-12345678`

A TEL-URI allows interaction with PSTN networks in which the clients are identified by a MSISDN. PSTN clients can also only use digits when dialing so it is important for an IMS Subscriber to (also) have a Public User Identity in the TEL-URI form.

The Service Profile is a collection of service and subscriber related data that can be used by Application Servers to allow more dynamic service logic. The Service Profile is stored on the HSS. An Application Server within the IMS network can (optional) access the HSS to retrieve this data. The Service Profile consists of Public Identification data, Authorization Data and Initial Filter Criteria.

3.4.5 Service Identity

A Service is identified by a Public Service Identity (PSI). The Public Service Identity is represented in the SIP URI or TEL-URI format. The PSIs of the services are stored in the HSS. A Service has also a Private Service Identity defined which is present for compatibility with the HSS interface.

3.4.6 Initial Filter Criteria

Each Public User Identity has a Service Profile. The Service Profile contains information on the Services that the subscriber is subscribed to. The list is of subscribed services is called the Initial Filter Criteria (IFC). Each IFC contains a reference to the Application Server and zero or more Service Trigger Points. The Service Trigger Points allows placing filters before a service is triggered. Supported filters are:

- Request-URI
- SIP Method
- SIP Header
- Session Case
- Session Description

If a service is triggered, the service is identified by its Public Service Identifier.

3.5 SIP AS

The SIP-AS is an IMS function accessible though the SIP interface that allows deployment of IMS Services. The IMS specification does not describe the actual implementation of a SIP-AS.

The SIP-AS can reside inside the operator's IMS network or outside the operator's IMS network. When the SIP-AS is located inside the IMS network the application server, an (optional) interface towards the HSS is defined using the Diameter protocol.

A SIP AS can operate (from a SIP perspective) as:

- Originating SIP User Agent, or User Agent Client (UAC)
  The SIP AS can create SIP requests.
- Terminating SIP User Agent, or User Agent Server (UAS)
  The SIP AS can receive and process SIP requests.

- SIP Proxy Server
  The stateless SIP AS routes SIP Message towards the destination.

- SIP Redirect Server
  The SIP AS

- SIP Back-to-Back User Agent (B2BUA)
  A statefull combined UAC and UAS that can apply application-specific logic.

The mode of operation of a SIP AS can change depending on the service being provided.

3.6 Examples

This chapter describes a number of use cases related to registration, setup and termination of a multimedia session and a simple IMS Service.

The examples describe sessions between SIP clients located in the users Home Network. That is, the user is not roaming in a Visitors Network.

3.6.1 Multimedia Session

3.6.1.1 Session Setup Example

The Session Setup use case describes the signaling during an IMS Session setup. The sequence is shown in Figure 8. The client is referred to as User Equipment (UE).

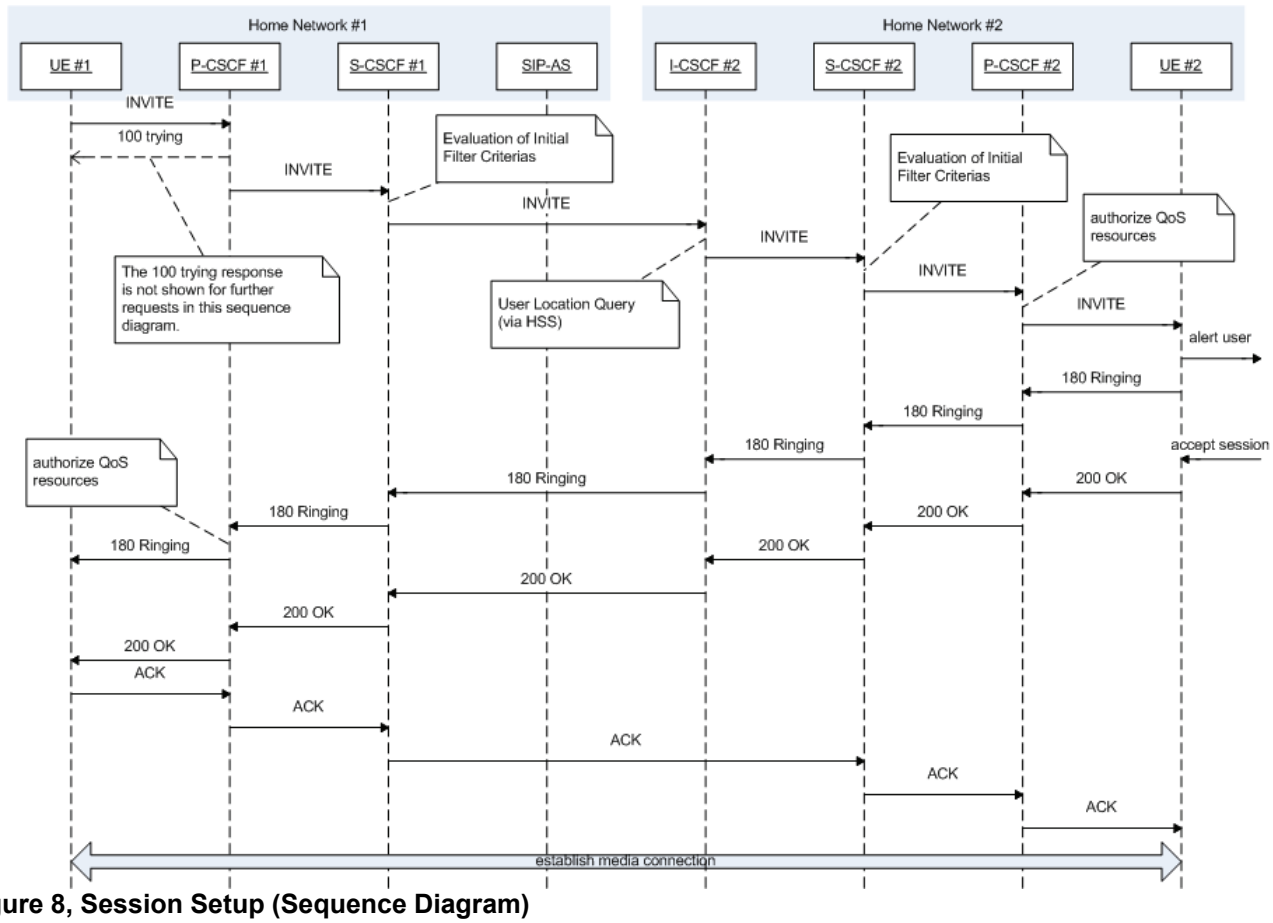


Figure 8, Session Setup (Sequence Diagram)

Basically, an INVITE request is send towards the UE #2. IMS subscriber 2 is alerted of the incoming session request. A preliminary response (180 Ringing) is returned. When IMS Subscriber 2 accepts the invitation a 200 OK response is returned (and acknowledged).

The S-CSCF #1 matches the INVITE request with the Initial Filter Criteria retrieved in the Session Profile of Subscriber 1 from the HSS. (In this use case there is no match.)

The INVITE is forwarded through the I-CSCF #2 towards the S-CSCF #2 where IMS Subscriber 2 is served. The S-CSCF #2 matches the INVITE request with the Initial Filter Criteria retrieved in the Session Profile of IMS Subscriber 2 from the HSS. (In this case there is no match.)

After the INVITE request is acknowledged, the Multimedia part of the IMS Session is setup.

3.6.2 Back To Back User Agent Service Example

The Back to Back User Agent Service example is similar to the Session Setup example with the exception that the INVITE A request now matches an Initial Filter Criteria. This is shown in Figure 9. Not all messages such as the 183 Ringing are shown.

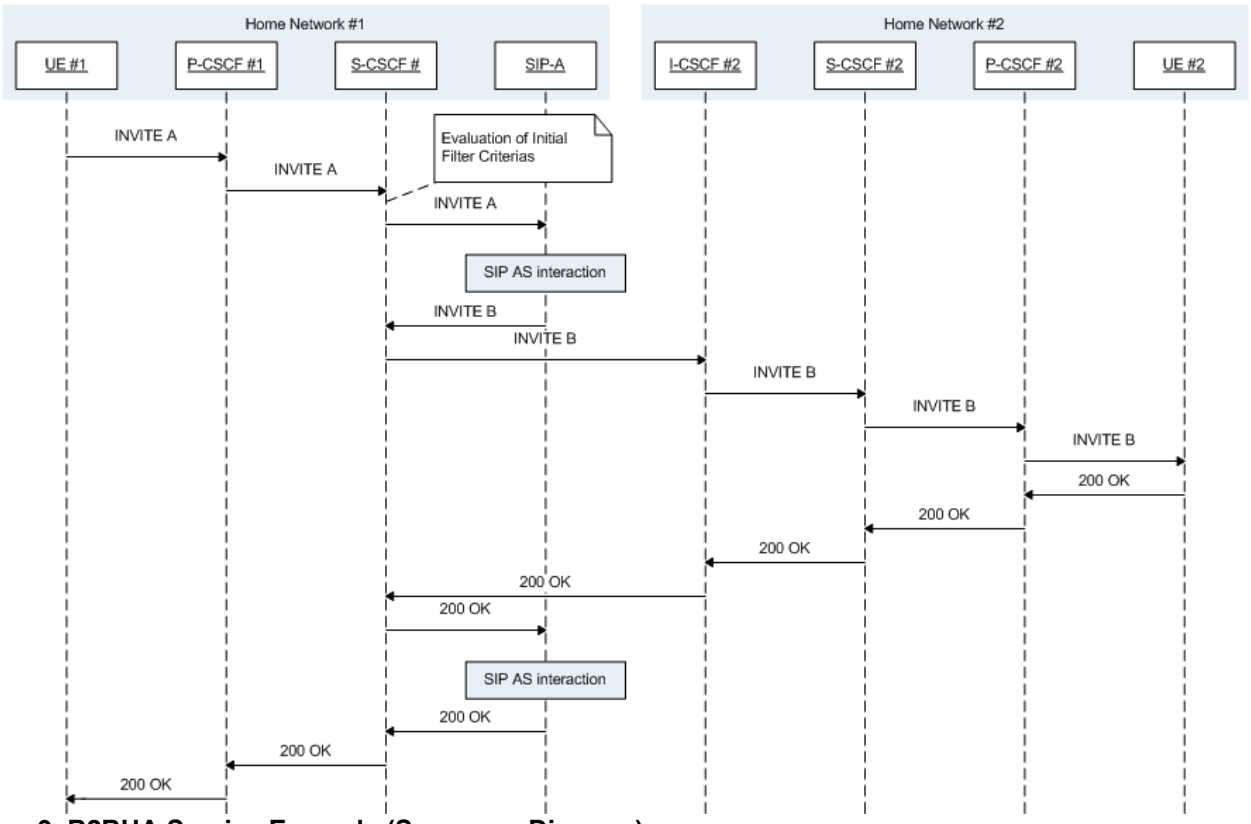


Figure 9, B2BUA Service Example (Sequence Diagram)

The S-CSCF #1 receives the INVITE A request of IMS Subscriber 1. This request is matched against the Initial Filter Criteria of Subscriber 1 and a match is found.

For example, an Initial Filter Criteria could contain a Trigger Point on all Originating SIP Messages with SIP Method INVITE. This would match all outgoing INVITE requests and not match all incoming INVITE requests.

Because of the match the INVITE A is routed towards the SIP-AS instead of the I-CSCF of IMS Subscriber 2. The SIP Application Server receives the request and can depending on the service act as a Proxy, Referrer, User Agent Server or Back to Back User Agent server. In the example the SIP-AS operates as a Back to back User Agent and creates the INVITE B request which is send to the S-CSCF #1. This INVITE B request does not match an Initial Filter Criteria and is routed towards the destination.

The 200 OK is routed back to the SIP-AS. The SIP-AS then creates a 200 OK which is routed towards the UE #1.

3.6.3 Multiple SIP AS Example

Multiple IMS Services can be configured and matched in an IMS Subscriber Service Profile. The services become part of the SIP Chain. An example with two SIP Application Servers is shown in Figure 10.

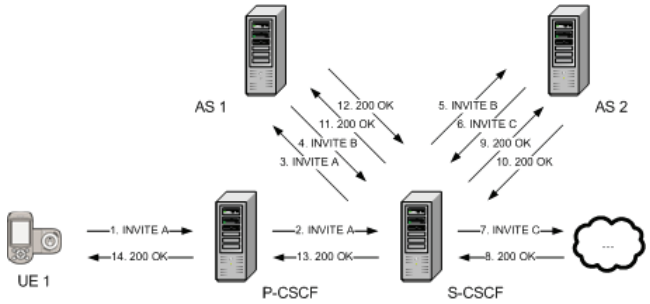


Figure 10, Session Utilizing Multiple SIP Application Servers

In this example the INVITE A is received by the S-CSCF and matches both the Initial Filter Criteria for AS 1 and AS 2. The Initial Filter Criteria for AS 1 has a higher priority and the INVITE A is routed towards AS 1. AS 1 sends request INVITE B towards the S-CSCF. The S-CSCF processes the next Initial Filter Criteria based on the priority and INVITE B is routed towards AS 2. AS 2 sends request INVITE C towards the S-CSCF. This INVITE C does not match an Initial Filter Criteria and the INVITE C is routed towards the destination.

3.7 SIP Container

A SIP Application Server can be implemented using a Sun Java Enterprise Edition Application Server with a SIP Container that provides a SIP stack and a framework to develop and deploy SIP Servlets.

The Java Community Process program provides two Java Specification Requests (JSR) that specify such an interface.

3.7.1 JSR-116

The Java Specification Request 116 (JSR-116) specifies the SIP Servlet Specification v1.0 [12]. JSR-116 provides a SIP stack and provides a framework to develop and deploy SIP Servlet based services. JSR-116 was released on March 7, 2003.

A SIP Servlet is an extension of a regular HTTP Servlet. The differences between a web service and a SIP service are listed below.

- A web service is always an origin server (which always generates the final response in a session). A SIP service does not always generate the final response in a session. A SIP service can also operate as a proxy server.
- A SIP service operates in a peer-to-peer session.
- A SIP service can originate requests
- A SIP service is asynchronous. It can return control to the container and initiate a (new) response request at a later time.
- The HTTP Servlet API selects a single HTTP Servlet. The SIP Servlet API can select multiple services in a SIP chain.

The JSR-116 SIP Container allows for converged applications in which both HTTP Servlet and SIP Servlet are supported.

The Sip Container provides:

- Servlet functionality (extends HTTP Servlet API)
- manage the life cycle of a session
- decide which Servlet to invoke and in which order
- a SIP stack for SIP traffic
- handling of timers, listener and event support
- handling of headers and routing

A service consists of one or more Servlets. Each service is described in the SIP deployment descriptor. This description includes a trigger which consists of (simple) rules. A trigger must match before a service is executed. Multiple services can match and are executed in the order in which they are listed in the SIP deployment descriptor.

The JSR-116 architecture is shown in Figure 11. Shown are the SIP Container that provides listen points towards the C-CSCF and three services. Service 1 and Service 2 are based on a SIP Servlet interface. Service 3 has an HTTP Servlet interface and illustrates the convergence between SIP and web services. A single Service can be accessed by multiple Servlets (not shown).

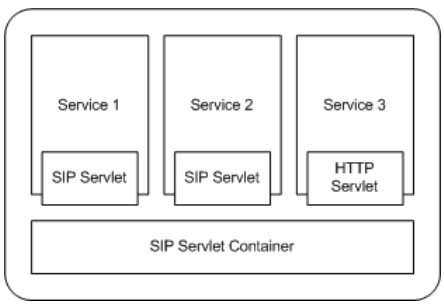


Figure 11, JS-116 Architecture

An example session is shown in Figure 12. This session describes an outgoing SIP dialog in which the INVITE has triggered a service on the S-CSCF. This service consists of Service A followed by Service B on the SIP Application Server.

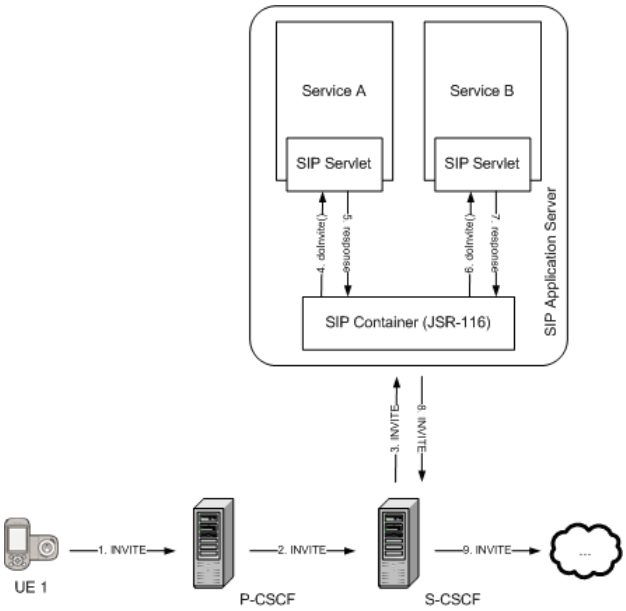


Figure 12, JSR-116 Session

The session is described below. The bold text refers to nodes or messages that are shown in the figure.

1 The IMS subscriber requests a multimedia session (e.g. places a call) using an IMS phone (**UE 1**). The UE 1 creates a SIP INVITE request (**1.INVITE**) which is delivered to the **S-CSCF** (**2. INVITE**).

2 The **S-CSCF** retrieves the session profile of the subscriber (not shown) and parses the Initial Filter Criteria (IFC). The IFC contains a matching Service Trigger Point. The **2. INVITE** request is send towards the application server indicated by the SIP URI present in the matching Service Trigger Point (**3. INVITE**).

3 The **SIP Container** receives the **3. INVITE** request and parses the services listed in the SIP deployment descriptor. The **3. INVITE** request matches two services: **Service A** and **Service B**. **Service A** is listed first in the SIP deployment descriptor and is executed first (**4. doInvite()**).

4 **Service A** is executed. In this example **Service A** operates as a proxy and does not create a new SIP request in the response (**5. response**).

5 The **SIP Container** receives the **5. response** and executes the second matching service listed in the SIP deployment descriptor (**6. doInvite()**).

6 **Service B** is executed. In this example **Service B** operates as a proxy and does not create a new SIP request in the response (**7. response**).

7 The **SIP Container** receives the **7. response** and returns the request towards the **S-CSCF** (**8. INVITE**) as there are no more matching services.

8 The **S-CSCF** receives the **8. INVITE** and forwards the request towards its destination (**9. INVITE**) as there are no more matching services in the Initial Filter Criteria.

In this example an initial request was configured in the subscriber's service profile and in the SIP deployment descriptor. This resulted in the Service A and Service B being added to the SIP chain. This chain remains and subsequent signaling messages will be routed through Service A and Service B.

3.7.2 JSR-289

The Java Specification Request 289 (JSR-289) [13] specifies the SIP Servlet Specification v1.1 and is an enhancement of JSR-116 (SIP Servlet Specification v1.0). JSR-289 is not yet released. An early draft was released on January 30, 2007.

JSR-289 adds the following enhancements:

- The logical entity Application Router
- Convergence with J2EE
- other enhancements

The JSR-289 architecture is shown in Figure 13. Shown are the SIP Container that provides listen points towards the C-CSCF and three services. Service 1 is based on a SIP Servlet interface. Service 2 is based on an HTTP Servlet interface. Service 3 illustrates the convergence with J2EE (through EJB and Web Services). The Application Router is a new logical entity.

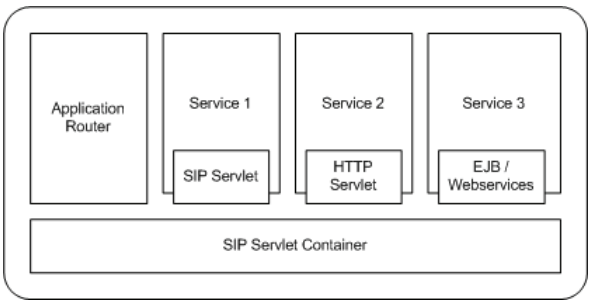


Figure 13, JS-289 Architecture

The Application Router decides what services to invoke and in which order through the Application Selection Process. The Application Router is only involved for the initial request. Subsequent requests are routed along the path created during the initial request.

The function of the Application Router is to provide service selection and should not contain application logic. To perform its role, the Application Router could access (private) data stores and could contain selection logic. A minimal implementation of the Application Router could be based on parsing a configuration file. The Application Router replaces the simple filter rules in the SIP deployment descriptor.

An example session is shown in Figure 14. This session describes an outgoing SIP dialog in which the INVITE has triggered a service on the S-CSCF. This service consists of Service A on the SIP Application Server.

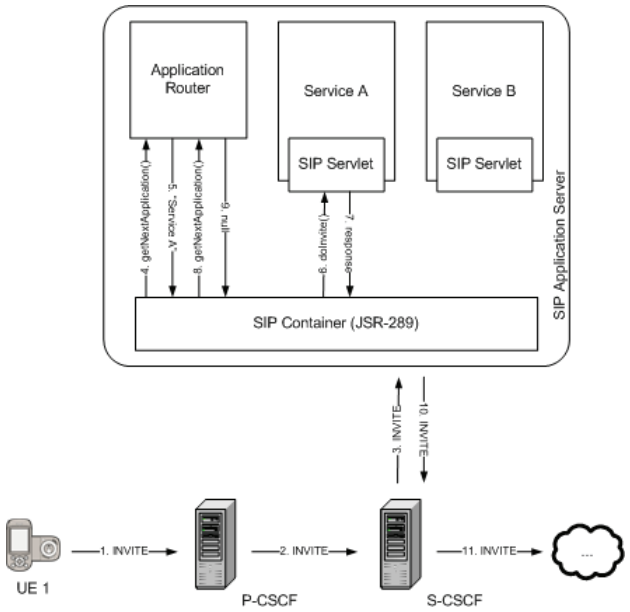


Figure 14, JSR-289 Session

The session is described below. The bold text refers to nodes or messages that are shown in the figure.

1 The IMS subscriber requests a multimedia session (e.g. places a call) using an IMS phone (**UE 1**). The UE 1 creates a SIP INVITE request (**1.INVITE**) which is delivered to the **S-CSCF** (**2. INVITE**).

2 The **S-CSCF** retrieves the session profile of the subscriber (not shown) and parses the Initial Filter Criteria (IFC). The IFC contains a matching Service Trigger Point. The **2. INVITE** request is send towards the application server indicated by the SIP URI present in the matching Service Trigger Point (**3. INVITE**).

3 The **SIP Container** receives the **3. INVITE** requests and queries the Application Router for the Service to execute (**4. getNextApplication()**).

4 The Application Router indicates **Service A** is the next service to execute. (**5. "Service A"**).

5 **Service A** is executed (**6. doInvite()**). In this example **Service A** operates as a proxy and does not create a new SIP request in the response (**7. response**).

6 The **SIP Container** receives the **7. response** and queries the Application Router for the Service to execute (**8. getNextApplication**).

7 The Application Router indicates there is no next service to execute (**9. null**)

8 The **SIP Container** returns the request towards the **S-CSCF** (**10. INVITE**) as there are no more services to execute.

9 The **S-CSCF** receives the **10. INVITE** and forwards the request towards its destination (**11. INVITE**) as there are no more matching services in the Initial Filter Criteria.

The Application Router is only invoked for initial requests. For subsequent request routing is based on the created SIP chain.

3.7.3 Project GlassFish and Project SailFin

GlassFish [14] is a free, open source Application Server implementing the SUN Java EE 5 standard.

SailFin [14] provides a JSR-289 compliant SIP Container to the GlassFish Application Server. Ericsson has contributed parts of its IMS server development to this project.

4 SCE

4.1 Introduction

At the start of the analysis of the SCE Service Model there was a lack of documentation on the SCE. This was remedied by creating a detailed description of the SCE Domain Specific Language used in the Composition Templates. This description provided a baseline for further analysis.

In addition, the XML Schema for the SCE Composition Template was not available and had to be reverse engineered.

The remainder of this chapter contains the description of the SCE Domain Specific Language.

4.2 Service Composition Environment

The Service Composition Environment (SCE) is an environment that enables design and execution of Telecommunication Services.

The SCE consists of an editor and an engine. The editor is used to design Composition Templates (the top level design artifact) and the engine is used to execute Composition Templates.

A Composition Template is created using a domain specific language visualized in the editor.

A Telecommunication Service consists of constituent services and the description of control and data flow between them. A constituent service is a runtime object that is executed by the engine during interpretation of the Composition Template.

The Composition Template indicates, through a number of constraints, which constituent service is required. Based on these constraints the engine decides during runtime which deployed executable object best matches these constraints.

The Service Composition Environment Domain Specific Language as described in this document is close to the actual implementation.

4.3 Terminology

Service Composition Environment (SCE)
: The Service Composition Environment consists of the Composition Engine and the Composition Design Environment.

Composition Engine
: The Composition Engine is the execution engine that interprets a Composition Template.

Composition Design Environment
: The Composition Design Environment is the design environment to create and edit Composition Templates.

Composition Template

A Composition Template is the top level design artifact that represents a Telecommunication Service. A Composition Template can be visualized using the Composition Design Environment and it can be interpreted by the Composition Engine.

Telecommunication Service

A Telecommunication Service (in the context of SCE) is a set of Constituent Services and a description of the control and data flow between them and could also be called a Composite Service. A Composite Service allows creating more complex services from less complex building blocks. This process is called Service Composition.

Constituent Service

A Constituent Service is a runtime executable object and is used as building blocks in a Composition Template. This could be a SIP Application or a Web Service.

4.4 Syntax

The SCE Domain Specific Language (DSL) is a visual representation of a Composite Template used in the Composition Design Environment.

4.4.1 Structure

The top level design artifact is a Composition Template. A Composition Template consists of Composition Template Elements (nodes) and Connections (edges). Composition Template Elements represent an action such as data control, flow control, or executing constituent services.

The following Composition Template Elements are supported:

- Start Element
- End Element
- Conditional Element
- Goto Element
- Service Element
- Composition Session Command Element

The Leave Container Service Element is a specialized form of a Service Element and is in this description considered as a distinct Composition Template Element.

Although not explicitly visualized, Composition Template Elements are considered to have Entry and Exit Points. The Entry and Exit Points are used to provide constraints on the number and direction of Connections.

Attribute can be “set” or “not set”. If an attribute is “set” this means the attribute has a value not equal to the empty string. If an attribute is “not set” this means the attribute has a value equal to the empty string.

The Composition Design Environment allows creation of Composition Templates for which the constraints are not met. These Composition Templates are considered to be invalid.

Connections and Composition Template Elements have attributes. Not all attributes may be visible on the Composition Template view in the Composition Design Environment. A separate Properties view is provided to inspect and change these attributes.

4.4.2 Connections

4.4.2.1 Description

A Connection represents a connection between two Composition Template Elements.

4.4.2.2 Syntax

A Connection is shown as an arrow between the Exit Point and Entry Point of two Composition Template Elements.

4.4.2.3 Attributes

- (String) Case

4.4.2.4 Constraints

- A Connection starts at a Composition Template Element Exit Point and ends at a Composition Template Element Entry Point.
- The Entry Point and Exit Point of a single Connection must not be on the same Composition Template Element.
- The Connection Case attribute must not be set for Connections starting not starting at a Conditional Element Exit Point.
- The Connection Case attribute may be set for Connections starting at a Conditional Element Exit Point. See description of the Conditional Element in chapter 4.4.6.
- All Composition Template Elements must be reachable (“connected”) from the Start Element. (Connections may not form loops.)
- All Entry and Exit Points must be connected.

4.4.3 Composition Template

4.4.3.1 Description

A Composition Template is the top level design artifact and represents a Telecommunication Service.

4.4.3.2 Syntax

A Composition Template consists of Composition Template Elements and Connections.

4.4.3.3 Attributes

- (String) Author
- (String) Composition Template constraints
- (String) Description
- (String) ID
- (Integer) Priority
- (String) Version

4.4.3.4 Constraints

- A Composition Template must have one and only one Start Element.
- A Composition Template must have at least one End Element.

4.4.4 Start Element

4.4.4.1 Description

The Start Element is a Composition Template Element.

4.4.4.2 Syntax

The Start Element is represented as a rectangle with a blue background.

The text inside the rectangle describes:

- The value of the ID attribute (in bold).
- The label “Composition Template:” and the value of the Composition Template ID attribute.
- The label “Priority:” and the value of the Composition Template Priority attribute.
- The label “Constraint:” and the value of the Composition Template Composition Template Constraints attribute.
- The label “Desc:” and the value of the Composition Template Description attribute.

An example of the Start Element is shown in Figure 15.

Figure 15, Start Element

4.4.4.3 Attributes

- (String) Description
- (String) ID
- (String) Type, has a fixed value “START_ELEMENT”.

The attributes of the Composition Template are linked to the Start Element.

4.4.4.4 Constraints

The Start Element has 0 Entry Points.

The Start Element has 1 Exit Point.

The Start Element Type attribute must be set.

The Start Element ID attribute must be set. (This must be a unique ID within the Composition Template.)

The Composition Template ID attribute must be set.

The Composition Template Priority attribute must be set.

4.4.5 End Element

4.4.5.1 Description

The End Element is a Composition Template Element.

4.4.5.2 Syntax

The End Element is represented as a rectangle with a red background.

The text inside the rectangle describes:

- The value of the ID attribute (in bold).

An example of the End Element is shown in Figure 16.

Figure 16, End Element

4.4.5.3 Attributes

- (String) Description
- (String) ID

- (String) Type, has a fixed value “END_ELEMENT”.

4.4.5.4 Constraints

The End Element has 1 Entry Point.

The End Element has 0 Exit Points.

The End Element ID attribute must be set. (This must be a unique ID within the Composition Template.)

4.4.6 Condition Element

4.4.6.1 Description

The Condition Element is a Composition Template Element.

4.4.6.2 Syntax

The Conditional Element is represented as a rectangle with a green background.

The text inside the rectangle describes:

- The value of the ID attribute (in bold).
- The value of the Condition attribute.

An example of the Condition Element, including three connections, is shown in Figure 17.

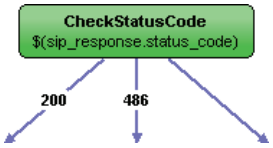


Figure 17, Condition Element

4.4.6.3 Attributes

- (String) Description
- (String) ID
- (String) Type, has a fixed value “CONDITION_ELEMENT”.
- (String) Condition

4.4.6.4 Constraints

- The Condition Element has 1 Entry Point.
- The Condition Element has 1 or more Exit Points.
- The Condition Element ID attribute must be set. (This must be a unique ID within the Composition Template.)

- The Connections connected to the Exit Points must have the Connection Case attribute set.
- Except for at most 1 connection that may have the Connection Case attribute not set.

4.4.7 Goto Element

4.4.7.1 Description

A Goto Element is a Composition Template Element.

4.4.7.2 Syntax

The Goto Element is represented as a rectangle with a yellow background.

The text inside the rectangle describes:

- The value of the ID attribute (in bold).
- The value of the Goto References attribute.

An example of the Goto Element is shown in Figure 18.

Figure 18, Goto Element

4.4.7.3 Attributes

- (String) Description
- (String) ID
- (String) Type, has a fixed value “CALL_ELEMENT”.
- (List of Goto References) Goto References

The type Goto References consists of:

- (String) Element ID
- (String) Composition Template ID

4.4.7.4 Constraints

- The Goto Element has 1 Entry Point
- The Goto Element has 1 Exit Point.
- The Goto Element ID attribute must be set. (This must be a unique ID within the Composition Template.)
- The Goto Element Goto Reference attribute must contain at least 1 Goto Reference.

4.4.8 Composition Session Command Element

4.4.8.1 Description

The Composition Session Command Element is Composition Template Element.

4.4.8.2 Syntax

The Composition Session Command Element is represented as a rectangle with a Grey background.

The text inside the rectangle describes:

- The value of the ID attribute (in bold).
- The label “Timeout:” and the value of the Timeout attribute.

An example of the Composition Session Command Element is shown in Figure 19.

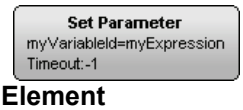


Figure 19, Composition Session Command Element

4.4.8.3 Attributes

- (String) Description
- (String) ID
- (String) Type, has a fixed value “SSM_COMMAND_ELEMENT”.
- (Command) Command
- (Command Parameter) Command Parameter
- (Integer) Timeout

The type Command is an enumerated type with values:

- (String) “undefined”
- (String) “setVariable”
- (String) “removeVariable”

The type Command Parameter consists of:

- (String) Variable ID
- (String) Expression

4.4.8.4 Constraints

- The Composition Session Command Element has 1 Entry Point.

- The Composition Session Command Element has 1 Exit Point.
- The Composition Session Command Element ID attribute must be set. (This must be a unique ID within the Composition Template.)
- The Composition Session Command Element Command attribute must be set to “setVariable” or “removeVariable”.
- The field Variable ID of the Composition Session Command Element Command Parameter attribute must be set.
- The field Expression of the Composition Session Command Element Command Parameter attribute must be set if the Command attribute is set to “setVariable” and must not be set if the Command attribute is set to “removeVariable”.

4.4.9 Service Element

4.4.9.1 Description

A Service Element is a Composition Template Element.

4.4.9.2 Syntax

The Service Element is represented as a rectangle with a turquoise background.

The text inside the rectangle describes:

- The value of the ID attribute (in bold).
- The label “Constraints” (in cursive).
- The value of the Constraints attribute.
- The label “Parameters” (in cursive).
- The value of the Parameters attribute.

An example of the Service Element is shown in Figure 20.

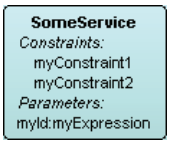


Figure 20, Service Element

4.4.9.3 Attributes

- (String) Description
- (String) ID
- (String) Type, has a fixed value “SERVICE_TEMPLATE_ELEMENT”.
- (Boolean) Asynchronous

- (List of Call Parameter) Call Parameters
- (List of Constraints) Constraints
- (String) Result Variable

The type Call Parameter consists of:

- (String) ID
- (String) condition

The type Constraint consists of:

- (String) Constraint

4.4.9.4 Constraints

- The Service Element has 1 Entry Point
- The Service Element has 1 Exit Point
- The Service Element ID attribute must be set. (This must be a unique ID within the Composition Template.)
- The Service Element Constraints attribute must contain at least 1 Constraint.
- The Constraints attribute must not contain the string “LeaveContainer” nor the string “dummy”.

4.4.10 Leave Container Service Element

4.4.10.1 Description

A Leave Container Service Element is a special configuration of a Service Element.

4.4.10.2 Syntax

The Leave Container Service Element is represented as a rectangle with a turquoise background.

The text inside the rectangle describes:

- The value of the ID attribute (in bold).
- The label “Constraints” (in cursive).
- The value of the Constraints attribute.

An example of the Leave Container Service Element is shown in Figure 20.

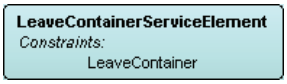


Figure 21, Service Element

4.4.10.3 Attributes

- (String) Description
- (String) ID
- (String) Type, has a fixed value “SERVICE_TEMPLATE_ELEMENT”.
- (Boolean) Asynchronous
- (List of Call Parameter) Call Parameters
- (List of Constraints) Constraints
- (String) Result Variable

The type Call Parameter consists of:

- (String) ID
- (String) condition

The type Constraint consists of:

- (String) Constraint

4.4.10.4 Constraints

- The Leave Container Service Element has 1 Entry Point
- The Leave Container Service Element has 1 Exit Point
- The Leave Container Service Element ID attribute must be set. (This must be a unique ID within the Composition Template.)
- The Leave Container Service Element Constraints must contain 1 Constraint.
- The Constraint must contain either the string “LeaveContainer” or the string “dummy”.
- The Leave Container Service Element Call Parameters attribute is not used and should not be set.

4.5 Semantics

4.5.1 Introduction

A Composition Template resembles a flow chart and consists of Composition Template Elements and connections between these elements. The execution follows a flow starting from the Start Element and, following the connections, ends at an End Element.

Certain Composition Template Elements may have as effect that the flow of the Composition Template is interrupted and an (external) constituent service is executed. The flow is resumed after the Composition Template is triggered by an event. This event is usually the result or response of the executed constituent service.

A constituent service is a building block of the Composition Template. Currently the following constituent services are supported:

- SIP Application
- Web Service

The constituent service is dynamically chosen during runtime by the Composition Engine based on the Constraints specified in the Composition Template Service Element.

A specialized form of the Service Element is the Leave Container Service Element. Execution of this element does not execute a constituent service. Instead the SIP message that initially triggered the Composition Engine is allowed to leave the SIP stack. This message is then routed by the network to its destination.

The Leave Container Service Element will interrupt execution of a Composition Template. Subsequent events such as SIP Responses will trigger the Composition Engine and execution of the Composition Template is then continued.

Execution of a Composition Template is started after the Composition Template is triggered by an Initial INVITE SIP Request event. Through a service selection process (not discussed in this document) a Composition Template is selected and execution starts at the Start Element.

The Composition Engine can handle multiple concurrent sessions.

This chapter provides an introduction on the SIP Chain, Event Types and Session. Next the semantics of the SCE DSL are discussed.

4.5.2 SIP Chain

The SIP Chain is an effect of the SIP protocol in which SIP Proxies and SIP Applications that are executed become part of the SIP Chain between the caller and callee.

The SIP Chain is build during routing of the SIP Initial INVITE Request. SIP Responses and subsequent SIP Requests will follow this chain. That is, the SIP Proxies and SIP Applications on the SIP Chain are notified of, and can act upon, these messages.

Any constituent services of type SIP Application that are executed by the Composition Engine will become part of the SIP Chain.

An example Composition Template that executes the constituent services A and B, both of type SIP Application, will result in a SIP chain that is shown in Figure 22.

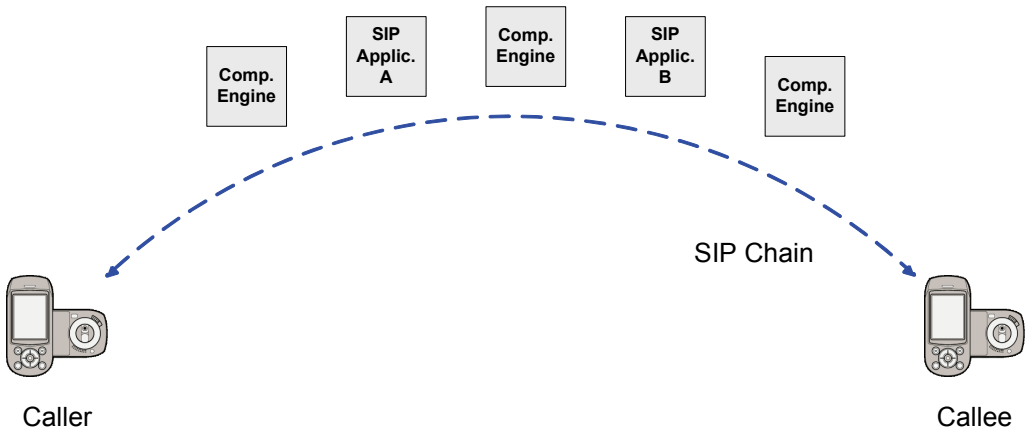


Figure 22, Example of SIP Chain

Constituent services of type Web Service are not related to SIP and do not become part of the SIP Chain.

Interesting consequences:

- All SIP Applications become part of the SIP Chain and will receive all subsequent SIP Messages independent of the Composition Engine.
- A single SIP Response or subsequent SIP Request will trigger the Composition Engine multiple times.

4.5.3 Events

The execution of a Composition Template is event driven.

SIP based events:

- SIP Initial INVITE Request
- Other SIP Requests
- SIP Provisional Responses
- SIP Final Responses

Web Service based events:

- Web Service Request
- Web Service Response

After receiving an event, the Composition Template Elements can access the event type and event properties through the Session.

The event that initially selects and starts a Composition Template is a SIP Initial INVITE Request.

4.5.3.1 SIP Based Events

Of the different SIP Request types only the initial INVITE SIP Request is currently supported by the Composition Engine. All other SIP Requests do not trigger the Composition Template and are processed using the default functionality of the SIP Container.

SIP Responses can be either Provisional responses or Final responses. SIP Responses are send in response to SIP Requests. Until a Final response is received, zero, one or more Provisional responses can be received. An example of a Provisional response is “180 Ringing”. An example of a Final response is “200 OK” or “486 Busy”.

Currently, neither Final nor Provisional SIP Responses trigger the Composition Template and all responses are processed using the default functionality of the SIP Container.

4.5.3.2 Web Service Based Events

The event Web Service Request is currently not supported by the Composition Engine.

The event Web Service Response is currently supported. After executing a Web Service, the Composition Engine will block until a Web Service Response is received.

4.5.4 Session

The Composition Engine keeps the state for existing sessions. This Session contains a parameter map consisting of:

- Session information (e.g. Composition Template information)
- Type and content of last received event
- Any stored variables

Variables can be stored in the Session using the Composition Session Command Element and can be retrieved using special keywords. For example: `$(sip_response.status_code)` for the Status Code if the last received event was a SIP Response.

4.5.5 Composition Template

The Composition Template Author attribute may contain a reference to the author of the Composition Template. This attribute has no effect on the execution of the Composition Template.

(String) Composition Template constraints

The Composition Template Description attribute may contain a descriptive text. This attribute has no effect on the execution of the Composition Template.

The Composition Template ID attribute must contain a unique (within SCE) identifier for the Composition Template. It can be used as a target Composition Template set in a Goto Element, but has no effect on the execution of the Composition Template otherwise.

The Composition Template Priority attribute is used during service selection. This attribute has no effect on the execution of the Composition Template.

4.5.6 Composition Template Elements

4.5.6.1 General

The Description attribute may contain a descriptive text. This attribute has no effect on the execution of the Composition Template Element.

The ID attribute must contain a unique (within the current Composition Template) identifier for the Composition Template Element. It can be used as a target Composition Template Element set in a Goto Element but has no effect on the execution of the Composition Template Element otherwise.

The Type attribute contains a value that specifies the type of the Composition Template Element. This attribute is automatically set and cannot be changed.

The directional Connection indicates the order in which the Composition Template Elements are executed. If a Connection is connected to the Exit Point of element A and the Entry Point of element B, then element B is executed after element A is executed.

4.5.6.2 Start Element

The Start Element represents the entry point of the Composition Template.

The Start Element performs no action. The flow continues with the execution of the Composition Template Element connected to the Exit Point.

The Start Element Priority attribute is used for Composition Template selection and does not influence the Composition Template execution.

The Start Element Constraint attribute is used to set global constraints used for constituent service selection.

4.5.6.3 End Element

The End Element represents the end of the Composition Template.

If the Goto Stack is not empty the Goto Element is popped from the Goto Stack and the Composition Template Element connected through to this Goto Element is executed next.

If the Goto Stack is empty the “leave container” action is returned. Any subsequent requests to the Composition Template are responded to with the “leave container” action.

4.5.6.4 Condition Element

The Condition Element decides, during runtime, through which of its Exit Points the flow will continue.

The Exit Point is selected using the following process:

1 Evaluate the expression given in the Condition Element Condition attribute.

2 Compare the evaluation result with the Case attribute of the Connections connected to the Exit Points of the Condition Element.

3 The first matching Exit Point is selected and the selection process ends.

4 If no Exit Points match and a default Connection is present, the Exit Point with the default Connection is selected. A default Connection is a Connection with an unset Case attribute.

The flow does not leave the container.

The Composition Template Element connected through the selected Connection is executed next.

4.5.6.5 Goto Element

The Goto Element continues the flow by executing a selected Composition Template Element than connected to the Exit Point of the Goto Element.

When the diverted flow executes an End Element, the flow continues with the execution of the Composition Template Element connected to the Exit Point of the Goto Element.

The Goto Element is pushed on a Goto Stack.

The selected Composition Template Element is configured by the Goto Element Goto References attribute.

4.5.6.6 Composition Session Command Element

The Composition Session Command Element allows to set a parameter in the Session or to clear a parameter in the Session.

If the Command attribute is set to “setVariable” the Expression field of the Command Parameter attribute is evaluated. The result is stored in the Session as the variable indicated by the Variable ID field of the Command Parameter attribute. Any existing variable in the Session is overwritten.

If the Command attribute is set to “removeVariable” the variable indicated by the Variable ID field of the Command Parameter attribute is cleared from the Session.

The flow continues with the execution of the Composition Template Element connected to the Exit Point of the Composition Session Command Element.

The Timeout attribute is not used.

4.5.6.7 Service Element

The Service Element allows execution of a constituent service. This constituent service is either a SIP Application or a Web Service and is decided during runtime by the Composition Engine based on the constraints set in the Service Element Constraints attribute and the constraints set in the Service Template Constraints attribute.

Parameters for the constituent service can be set using the Service Element Parameters attribute.

Execution of a constituent service will stop the flow of the Composition Template.

After an event is received, the flow will continue with the execution of the Composition Template Element connected to the Exit Point.

4.5.6.8 Leave Container Service Element

The Leave Container Service Element is a special type of Service Element. The Leave Container Service Element has no Parameters attribute set and the Constraints attribute must contain the value “LeaveContainer” or “dummy”.

The Composition Engine takes no action. That is, any SIP Message that triggered the Composition Template is returned to the container.

This will stop the flow of the Composition Template.

After an event is received, the flow will continue with the execution of the Composition Template Element connected to the Exit Point.

4.6 Examples

4.6.1 Example 1

The first example shows a Composition Template that executes two constituent services. The Start Element Constraint attribute and the two Service Element Constraints attributes are chosen in such a way that a SIP Application is executed during runtime.

The Composition Template for Example 1 is shown in Figure 23.

This Composition Template will form a SIP Chain as indicated in Figure 22.

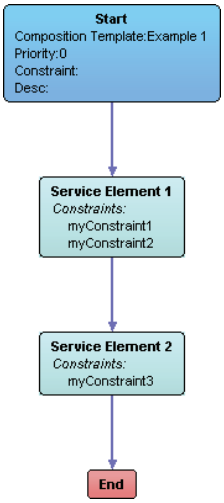


Figure 23, Composition Template Example 1

The execution of the Composition Template is discussed below:

1 Caller setups a call towards Callee. This creates an Initial INVITE SIP Request that is routed towards the network.

2 The Initial INVITE SIP Request is intercepted by the network and is forwarded towards the Composition Engine.

3 In the Composition Engine the Initial INVITE SIP Request triggers execution of the “Example 1” Composition Template.

4 The Start Element is executed. No action is associated with a Start Element and the flow continues with the Composition Template Element connected to the Exit Point of the Start Element.

5 The “SIP Application A” Service Element is executed. Based on the Constraints attribute a decision is taken during runtime which constituent service is executed. In this example “SIP Application A” is executed.

6 The Initial INVITE SIP Request is redirected towards “SIP Application A” and execution of the Composition Template is interrupted.

7 “SIP Application A” receives the Initial INVITE SIP Request and performs some action. After “SIP Application A” is finished the Initial INVITE SIP Request is send back towards the Composition Engine.

8 An Initial INVITE SIP Request is received and the Session is retrieved. The Composition Template is triggered and the flow continues with the Composition Template Element connected to the Exit Point of the “SIP Application A” Service Element.

9 Steps 5 through 8 are repeated but for “SIP Application B”.

10 The End Element is executed. A “Leave Container” action is associated with this Composition Template Element. The Initial INVITE SIP Request is returned to the network and is forwarded towards the Callee.

4.6.2 Example 2

The second example shows a Composition Template that provides a Call Forwarding on Busy service.

If caller calls callee and callee is busy, the Call Forwarding on Busy service automatically forwards the call towards a forwarded user.

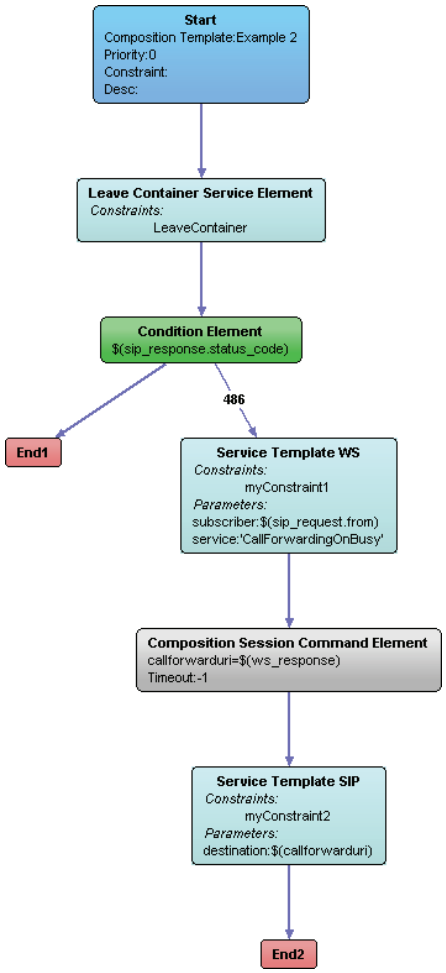


Figure 24, Composition Template Example 2 - Call Forwarding On Busy

The execution of the Composition Template is discussed below:

1 Caller setups a call towards Callee. This creates an Initial INVITE SIP Request that is routed towards the network.

2 The Initial INVITE SIP Request is intercepted by the network and is forwarded towards the Composition Engine.

3 In the Composition Engine the Initial INVITE SIP Request triggers execution of the “Example 1” Composition Template.

4 The Start Element is executed. No action is associated with a Start Element and the flow continues with the Composition Template Element connected to the Exit Point of the Start Element.

5 The “Leave Container” Service Element is executed. The Initial INVITE SIP Request is returned to the network and is forwarded to Callee. Execution of the Composition Template is interrupted.

6 If Callee answers the call a “200 OK” SIP Response is send back over the SIP Chain. If Callee is busy, a “486 Busy” SIP Response is send back.

7 The SIP Response is received by the Composition Engine and the Session is retrieved. The Composition Template is triggered and the flow continues with the Composition Template Element connected to the Exit Point of the “Leave Container” Service Element.

8 The Condition Element is executed. The expression in the Condition attribute is parsed. This returns the status code of the SIP Response. The Condition Element has two Exit Points. The first Connection has the Case attribute set to “486”, the second Connection has the Case attribute not set.

9 If the Callee answered the call a “200 OK” Sip Response was send back. The Status Code (the string “200”) does not match any Connections and the default Connection is followed. This would result in the execution of the End Element, which would end the Composition Template and the call would continue to be setup between the Caller and Callee.

10 If the Callee answered with busy a “486 Busy” SIP Response was send back. The Status Code (the string “486”) matches one of the Connections. This would result in the execution of the “Service Template WS” Service Template.

11 The “Service Template WS” represents a constituent service of type Web Service. A subscriber and service parameter are included. The Web Service returns the address used to forward the call. Executing the Web Service will interrupt execution of the Composition Template.

12 The Web Service is executed and a Web Service Response is received by the Composition Engine and the Session is retrieved. The Composition Template is triggered and the flow continues.

13 The Composition Session Command Element stores the Web Service result in the Session.

14 The “Service Template SIP” represents a constituent service of type SIP Application. A destination parameter is included. The SIP Application redirects the call towards the address retrieved by the Web Service. Executing the SIP Application will interrupt execution of the Composition Template.

15 On any event received (but this would typically be a SIP Response from the Forwarded Address) the Composition Template continues with the End Element, which will return the event to the network and end the execution of the Composition Template.

Note: The current implementation of the Composition Engine has several limitations:

- SIP Responses are not handled
- Parameters are not supported for Web Services
- Parameters in the format used in the example are not supported by SIP Applications.

The Composition Template Example 2 is therefore given as an example only. It can currently not be executed.

4.6.3 Serializable Format

4.6.3.1 Introduction

The SCE DSL can be visualized using the Composition Design Environment. The Composition Design Environment also allows exporting and importing a Composition Template to and from the file system.

4.6.3.2 XML Schema

The file format used is based on XML and is described by an XML Schema (see Appendix A).

5 CFB Service

5.1 Introduction

A simple communication service was chosen to use as an example throughout the investigation of a service model.

This *Call Forwarding on Busy* (CFB) service is a small and simple service that still contains a number of interesting features.

The Call Forwarding on Busy is part of a collection of Communication Diversion (CDIV) services [4]. The set of CDIV services itself is part of the Multimedia Telephony (MTEL) set of services [3] for IMS that simulate "circuit switched" services.

This chapter describes the CFB service as used throughout the investigation. A reference implementation and several models of the CFB service are given.

5.2 Call Forwarding on Busy Analysis

5.2.1 Service Behavior

The behavior of the CFB service is described below.

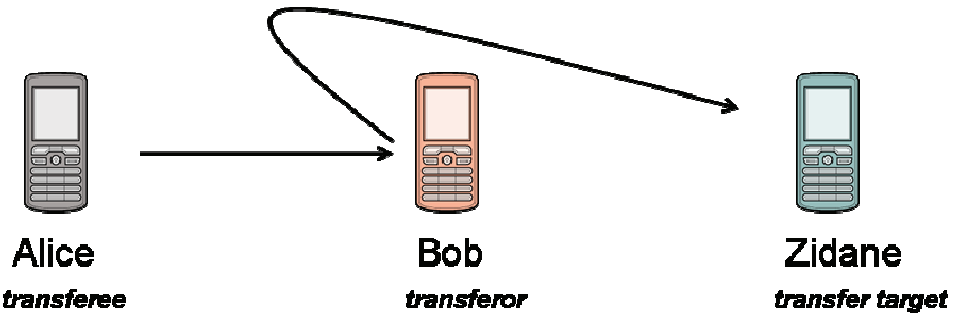


Figure 25, Call Forwarding on Busy Service

Alice calls Bob. If Bob has the CFB service enabled and Bob responds with BUSY, Alice receives a notification that the call is being forwarded, and the call is forwarded to Zidane. The service remains idle if Bob answers with anything else then BUSY.

The session flow of the Call Forwarding on Busy Service is shown in Figure 25.

The name "Zidane" is given as an example. Bob would be able to specify the transfer target in his user profile. Bob would also be able to active or de-activate the CFB service. Such a user profile and the interface towards it are outside the scope of the CFB service. In the example implementations a hard coded reference to "Zidane" is used.

The Call Forwarding on Busy service as used as an example service during the investigation is a simplified version of the service described in the CDIV specification. For example, the CDIV describes the use of additional SIP Message header fields to keep a history of forwards performed earlier during the setup of the SIP session. These fields prevent looping when, for example, Zidane is busy and forwards the call to Bob. Setting and checking of these fields is not included in the example implementations of the CFB service.

5.2.2 Service Specification and Implementation Choices

The TISPAN CDIV specification describes the behavior of the Call Forwarding on Busy service, but does not describe the implementation for this service. It was found the CFB service could be implemented in a several ways.

- Using a "302 Call is Being Redirected" SIP Response send by the *transferor*. Note that this actually bypasses the SIP Application Server.
- Using a "302 Call is Being Redirected" SIP Response send by the SIP Application Server. The CFB Service operates as a SIP Proxy.
- By operating as a SIP Proxy and intercepting the SIP Messages on the SIP dialog between the parties.
- By operating as a SIP B2BUA and intercepting the SIP Messages on the SIP dialog between the parties.

The third and fourth options most closely match an informative example given in [4]. Because a Proxy implementation is less intrusive than a B2BUA implementation, the design of the reference CFB service implementation was based on the third option.

5.3 Call Forwarding on Busy as SIP Proxy

The Call Forwarding on Busy service implemented as a SIP Proxy was further analyzed. This resulted in the following artifacts.

- A Sequence Diagram
- A reference implementation based on JSR-116
- A reference implementation based on SCE
- A State Chart Diagram

The purpose of these implementation and modeling exercises was to gain an understanding of the service and to create a baseline for further analysis.

5.3.1 Sequence Diagram

A set of sequence diagrams was constructed to describe the exact behavior of the Call Forwarding on Busy service. These sequence diagrams were constructed based on the IMS and MTEL / CDIV specifications.

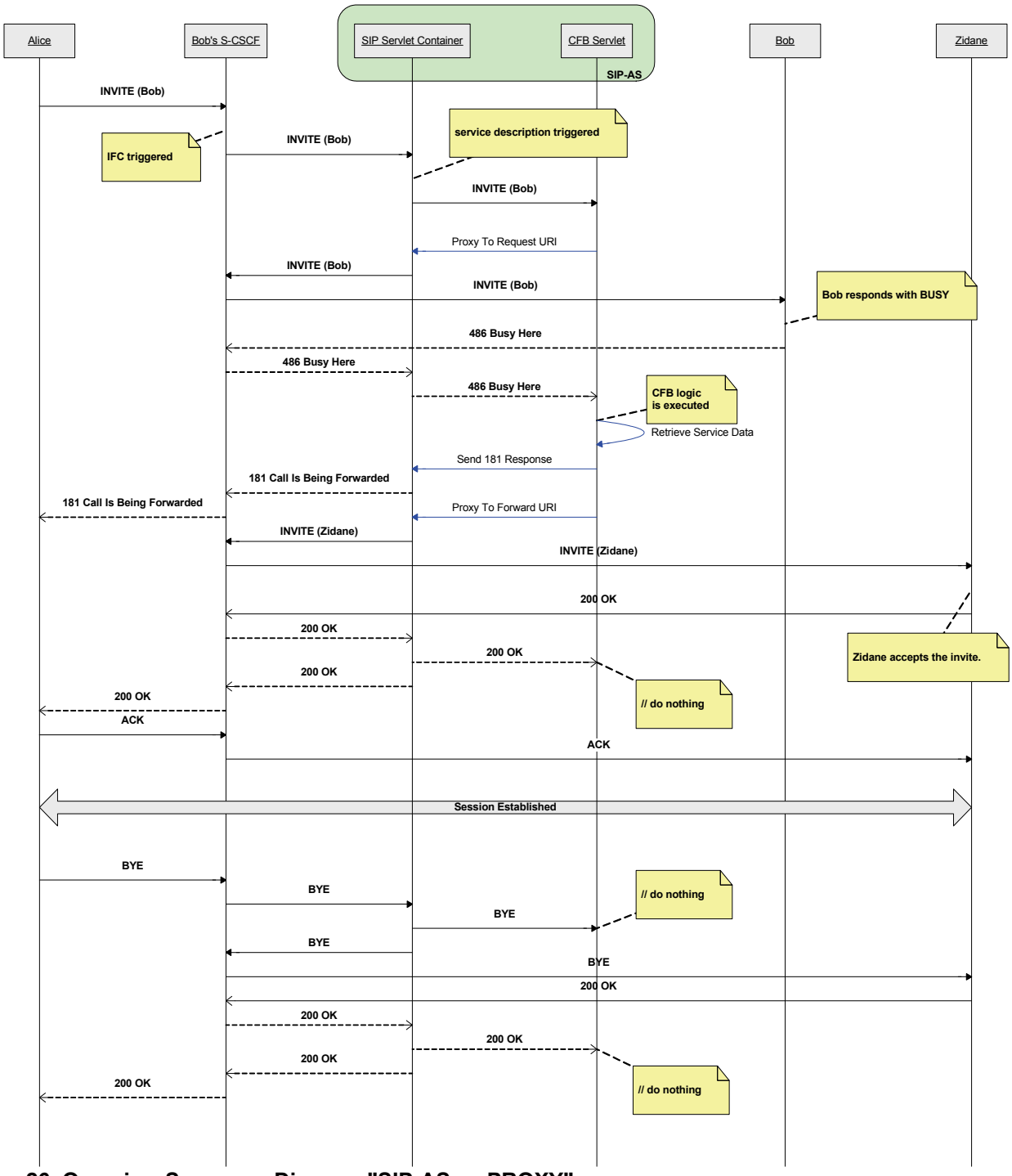


Figure 26, Overview Sequence Diagram "SIP-AS as PROXY"

The sequence diagram shown in Figure 26 provides a (simplified) overview of the CFG service. The following remarks are clearly visible in the diagram.

- The SIP Application (both the SIP Container and the CFB Servlet) become part of the SIP session. This can be seen by following the `INVITE (Bob)` messages.
- The CFB Servlet remains on the SIP session (and SIP Chain) through the `Proxy To Request URI` action.
- The busy response is received by the CFB Servlet and the CFB logic is executed. This involves:

- dropping the busy response,
- responding with a `181 Call is Being Forwarded` response towards Alice,
- sending an invite towards Zidane.

- A "do nothing" action is often used by a SIP Application. This action is also used several times by the CFB service.

5.3.2 SIP Application

A reference implementation based on the JSR-116 SIP Container was implemented as a proof of concept and as a baseline for further developments. The SDS design environment was used. This design environment includes simulations of IMS nodes including a JSR-116 compliant SIP Application Server.

Fragments of the reference implementation are shown in Figure 27. The CFB logic is contained in just a few lines of code, shown as bold.

```
/**
 * doInvite method called when SIP INVITE Request is received.
 */
protected void doInvite(SipServletRequest sipServletRequest)
throws ServletException, IOException
{
        // Proxy to request URI
        Proxy proxy = sipServletRequest.getProxy();
        proxy.proxyTo(sipServletRequest.getRequestURI());
}

/**
 * doErrorResponse method called when 4xx SIP Response is received.
 */
protected void doErrorResponse(SipServletResponse resp)
throws ServletException, IOException
{
        // CFB is only for BUSY
        if (resp.getStatus() == SipServletResponse.SC_BUSY_HERE)
        {
                // Retrieve Service Data (hardcoded in this example)
                SipURI forwardUri = sipFactory.createSipURI("zidane", "ericsson.com");

                // Send 181 Call Being Forwarded provisional response
                SipServletRequest sipServletRequest = resp.getRequest();
                SipServletResponse provisionalResponse =
                        sipServletRequest.createResponse(SipServletResponse.SC_CALL_BEING_FORWARDED);
                provisionalResponse.send();

                // Proxy to forwarded URI
                Proxy proxy = resp.getProxy();
                proxy.proxyTo(forwardUri);

                return;
        }

        // default behaviour
        super.doErrorResponse(resp);
}
```

Figure 27, CFB SIP Application

The purpose of this implementation is to:

- Serve as a proof of concept. The CFB service can be implemented as a SIP Application.
- Act as a reference implementation for future Composition Template or State Machine based implementations.

- Learn how to implement a (simple) SIP Application. SIP Development differs from Circuit Switched Telecommunication Service frameworks or HTTP Servlet based services.

Approach:

- Study the CFB specification. Study the SIP Container documentation.
- Using the Ericsson SDS design environment, develop the CFB service.
- Using the Ericsson SDS design environment, test the CFB implementation.

Observations:

- The process to design, implement and test the CFB SIP Application took considerable time.
- No reference implementations were readily available. Out of several design directions the best reference implementation needed to be chosen. The given reference CFB SIP Application is chosen for its simplicity. An alternative (and more powerful) implementation could be based on B2BUA instead of the (simple) Proxy based implementation.
- Although specifically mentioned in the JSR-116 specification, the generation of a provisional 181 Call Being Forwarded response generated a runtime error. This was initially believed to be caused by a development error in the CFB implementation. Later it was found to be caused by the used implementation of the JSR 116 SIP Container. After switching from the SDS Sip Container to a BEA Weblogic SIP Container the problems were solved. It appears that this feature is not correctly implemented in various JSR-116 libraries.
- The SIP Chain model and the supporting SIP Container are rather powerful. A simple CFB service is implemented using a relative small amount of code.

The SDS design environment includes IMS simulators as well as testing tools. These tools were used to perform end to end testing of the CFG service through the use of "Alice", "Bob" and "Zidane" simulated clients.

5.3.3 State Chart Diagram

In addition to the Sequence diagrams, a number of State Chart diagrams were created to represent the behavior of the Call Forwarding on Busy service.

The first effort, shown in Figure 28, was closely based on the states defined by the SIP protocol: Setup, Early, Confirmed and Terminated.

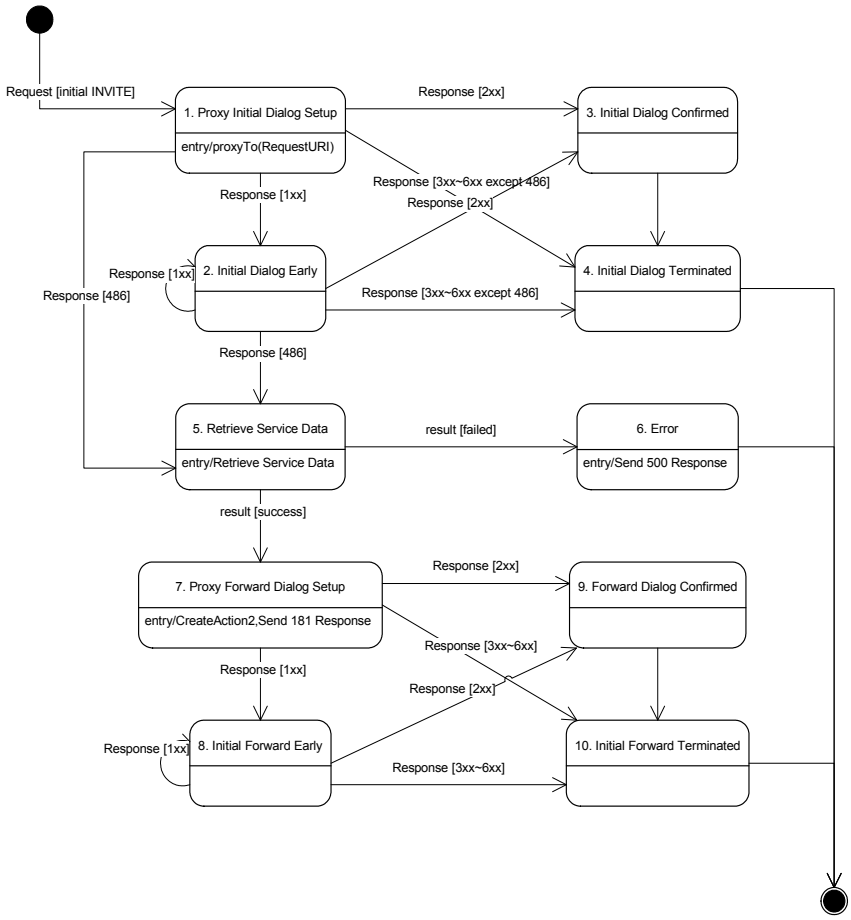


Figure 28, State Chart CFB

The mapping onto the SIP states was found not to add any value to the model. In contrast, the model was too large.

An updated version of the State Chart diagram was designed with the purpose to create a concise model in which the CFB service was clearly visible. This alternate model is shown in Figure 29. The number of states is reduced to five (the Initial and Final states are counted as well). While the mappings to the SIP states are dropped, the model still represents a valid SIP service.

Corresponding State Tables for the diagrams were also created, but these were never used.

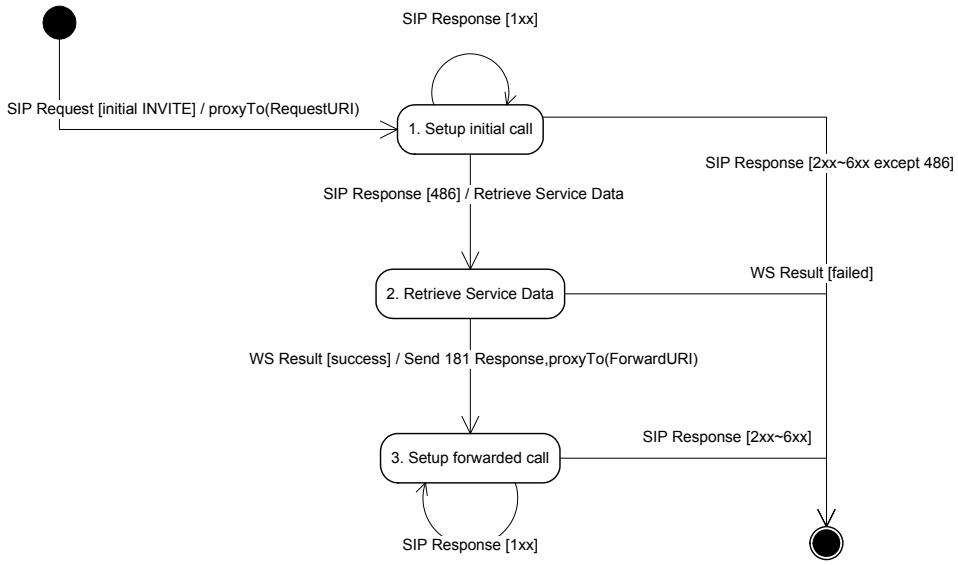


Figure 29, State Chart CFB (alternative)

The representation of actions in the transitions is chosen as the actions are closely related to the events. When the actions are specified as pre and post actions in the states itself, this relations is not clear. Only including actions in transitions also allows specifying a “no action” event such as a Provisional SIP Response.

SIP Requests, except for the initial INVITE, are not handled in the State Chart. An example would be the CANCEL SIP Request. Note that the SIP Container has a default handling of certain SIP Messages such as ACK, CANCEL. In addition, only events that are expected are present. E.g. If a SIP Response is expected, no WS Result is indicated and visa versa.

There is a difference in concept regarding the final state. From a model perspective, the final state is final. No further transitions are possible. From the SIP service perspective, the "END" state implies the service remains idle. That is, the service can still receive triggers, but will perform the "do nothing" action. Termination of the service is up to the SIP Container.

Figure 30, Final State vs. Simple State

It could be argued that the correct representation would be to replace the final state in the model with a simple state that contains a transition looping back to the simple state. This is demonstrated in Figure 30. For simplicity reasons, the use of a final state was retained.

6 SCE State Machine Transformation Specification

6.1 Introduction

This chapter provides a description of the transformation from a UML State Chart to a Composition Template using the XSLT language.

An overview of the observations and design choices made is provided in chapter 7.

6.2 XSLT

XSLT stands for XSL Transformations and is a language for transformation of XML documents. XSL (eXtensible Stylesheet Language) is a collection of transformation languages.

XSLT is a W3C Recommendation [5].

The transformation is based on XSLT Version 1.0.

6.3 UML State Chart

A UML State Chart is the source of the transformation. It is assumed this UML State Chart represents a valid Telecommunication Service and conforms to the Design Guidelines (see chapter 6.3.1). These guidelines define the structure of the State Chart as used by the transformation.

Unified Modeling Language (UML) [7] is an object modeling language specified by the Object Management Group (OMG).

The OMG XML Metadata Interchange (XMI) [8] standard is used for serializing the UML State Chart.

The transformation is based on UML Version 1.4 and XMI Version 1.2.

The terms *transition*, *state*, *event*, *trigger*, *guard*, *effect* and *action* are used as defined in the UML specification.

6.3.1 State Chart Model Design Guidelines

The Design Guidelines are listed below. Basically, they define a Mealy State Machine with actions only defined for transitions.

- The State Chart Model only supports a single top Complex State that represents the State Chart. No further Complex States are allowed.
- The Model must define one Pseudo State of kind initial. No (deep) history, join, choice, etc are supported.
- The Model must define zero or more Simple States.
- The Model must define one or more Final States.

- The initial vertex (e.g. the start element) has 1 outgoing transition with a trigger of type SIP_Request and a guard with value INVITE. Zero, one or more effects may be specified.
- The state name values must be unique and not empty.
- All states must be reachable. A final state should be reachable from each simple state.
- Each transition must have a trigger and guard set.
- States may not have a trigger, guard or action set.
- A state is identified by its name property. This value must be a valid SCE element name.
- A trigger indicates an event. An event is always of type CallEvent and is identified by its name property. This value must be either the keyword `WS_RESPONSE`, `SIP_REQUEST` or `SIP_RESPONSE`.
- A guard is set through its expression property. The value should conform to the SCE syntax in which the keyword `status` can be used. Example: `(100 <= status) && (status < 200)`
- For a given state and trigger type, the guards may not overlap.
- For a given state and trigger type, the guards must cover the complete domain. E.g. all valid status values between 100 and 699.
- An action is either a single CallAction or an ActionSequence that contains CallActions. A CallAction is set through its name property.
- An action that represents a SIP constituent service must be part of an ActionSequence in which the last action represents the Leave Container SIP constituent service.
- The modeled service must be based on SIP. E.g. the state transitions must match the transitions that are allowed and possible in SIP. Also, actions (concerning session routing/SIP) must represent valid SIP actions such as proxy, send provisional response, or cancel.
- The actions used in the state chart are eventually mapped to service template elements of a composition template. The description of actions should be chosen in such a way that this mapping is possible.

A hierarchical structure of a state chart is depicted in Figure 31.

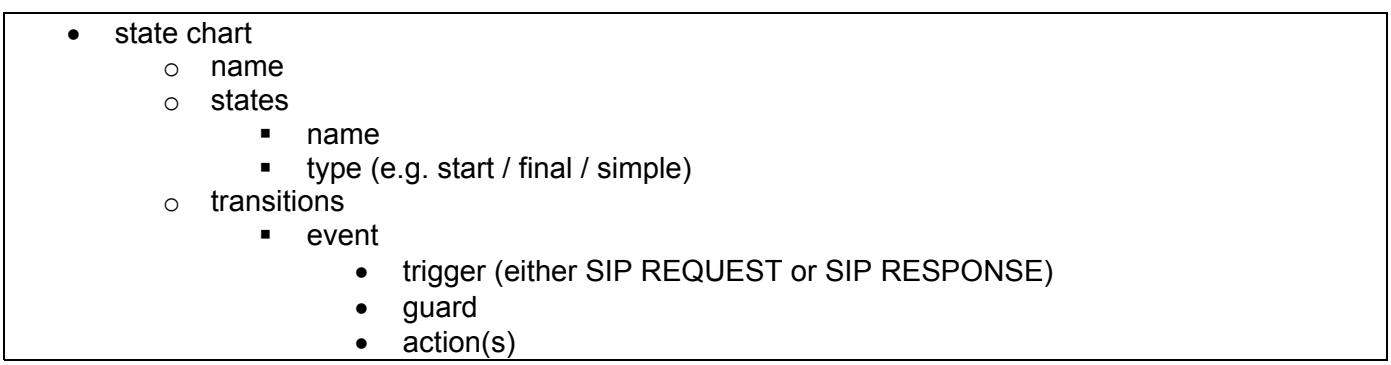


Figure 31, Overview Generic State Chart Structure

6.4 SCE Composition Template

The Service Composition Environment (SCE) is an environment that enables design and execution of Telecommunication Services. A Telecommunication Service is represented by a Composition Template (the top level design artifact).

The Service Composition Environment and Composition Templates are described in chapter 4.

A Composition Template can be exported and imported as a XML Document which is described by a XML Schema document.

The SCE DSL contains the following features that are at the basis of a State Machine implementation:

- a session (to store the current state)
- a conditional element (to evaluate the current state, trigger, or guard and branch depending on the result)
- a service element (to execute effects)
- a goto element (to handle the next event)

6.5 Two Step Transformation

We selected a two step approach for the transformation.

The first step transforms the UML State Chart document into an Intermediate XML document using a concise format. This removes complex XSLT constructions and XPath expressions from the actual XSLT description.

The second step in the end to end transformation performs the actual transformation into a Composition Template document.

Each transformation is specified by a XSLT document.

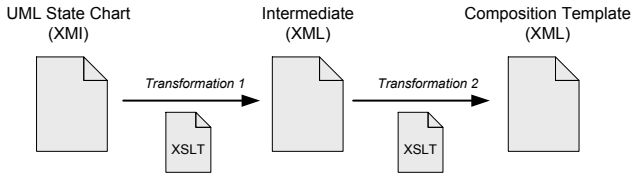


Figure 32, Two Step Transformation

The remainder of this chapter describes the second transformation; the transformation from the Intermediate document into a Composition Template document.

6.5.1 State Machine Basic Structure

The State Machine implementation as a Composition Template is based on the following general flow:

1 Evaluate the state parameter stored in the session. Continue the execution flow with the branch matching the value of the state parameter.

2 Evaluate the trigger of the event. Continue the execution flow with the branch matching the trigger.

3 Evaluate the guard of the event. Continue the execution flow with the branch matching the guard.

4 Execute the effects of the event (if any).

5 Update the state parameter stored in the session based on the transition caused by the event.

6 Continue execution with step 1.

The State Machine implementation contains an initialization.

- An initializing part that stores a reference to the initial state in the state parameter. This part is executed during startup of the Composition Template and is performed only once.

Final states are processed differently.

- One or more End elements that represent Final State(s). An End Element could be reached as a result of Step 1. In this case, steps 2 through 6 are not executed.

The basic structure of the State Machine is shown in Figure 33.

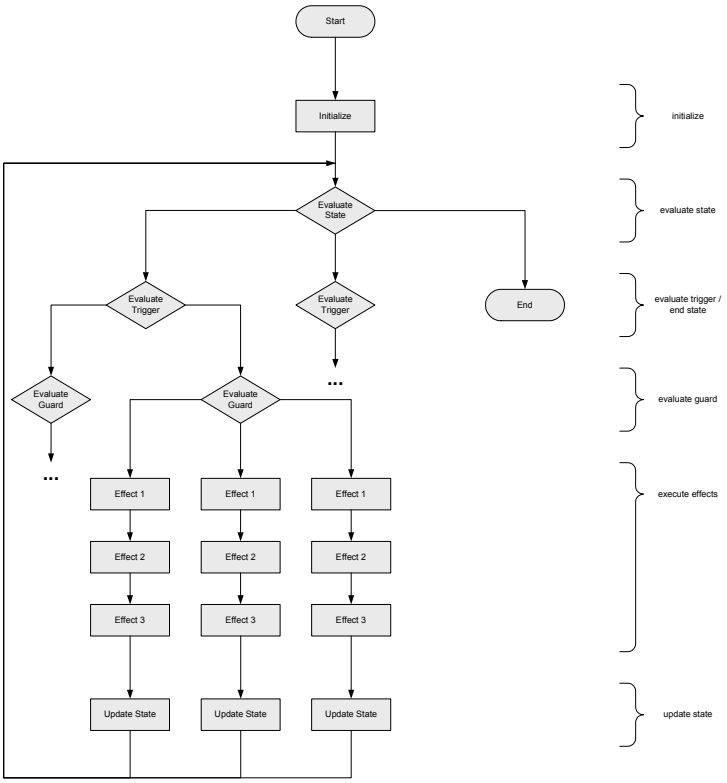


Figure 33, Flow Diagram of State Machine implementation

6.5.2 XSLT Basic Structure

See Appendix A for a reference of the XSLT documents described in this chapter.

6.5.2.1 Declarative

XSLT is a declarative language. This means the language describes the target document and not the process to create the target document.

6.5.2.2 Template Rules and Named Templates

The XSLT transformation is based on a Template Rule and a number of Named Templates.

A Template Rule consists of a pattern and a template. The pattern is an XPath expression that is executed on the current node of the source document. In the transformation the Template Rule is used to select the root element of the source document. Matching the root element is the first step of XSLT processing. This adds the template (the contents of the `<template>` element) to the target document. The template may contain instructions that are processed further. The transformation uses the `<call-template>` instruction, which references Named Templates.

Named Templates consist of a name and a template. Named Templates are processed when invoked (by name) by the `<call-template>` instruction.

The transformation is based on a Template Rule that matches on the root. Processing this Template Rule adds the content for an empty Composition Template to the target document. Named Templates are then used hierarchically to add the Composition Template Elements and Connections to the target document.

Table 2 describes the Named Templates and provides a description of their function.

Named Template	**Description**
`process`	Processes the `process_generic` Named Template. Parses through all states and processes the `process_startstate_element`, `process_endstate_element` or `process_simplestate_elements` Named Templates based on the state type.
`process_generic`	Adds the Composition Template Elements (and Connections) that are not dependent on a state. For example, these are the Start Element, and the Conditional Element that evaluates the state.
`process_startstate_element`	Adds the Composition Session Command Element that stores the initial state. Processes the `process_simplestate_elements` Named Template for the given state.
`process_endstate_elements`	Adds an End Element (and Connection) for the given end state.
`process_simplestate_elements`	Adds, depending on the trigger type, the Conditional Elements and Connections as part of the evaluate trigger step. Adds a Connection to the created Conditional Element as part of the evaluate state step. Processes, depending on the trigger type, the

	process_single_vertex_transitions or process_multi_vertex_transitions Named Template.
process_single_vertex_transitions	Adds, depending on the guard type, Conditional Elements and Connections as part of the evaluate guard step. Adds a Connection to the created Conditional Element as part of the evaluate trigger step. Processes, the process_effects Named Template.
process_multi_vertex_transitions	The process_multi_vertex_transitions Named Template is an alternative of the process_single_vertext_transitions and uses a different construction of the Conditional Elements that perform the evaluate guard step.
process_effects	Adds, if effects are available, the Service Elements (and Connections) that represent the effects. Adds the Composition Session Command Element (and Connections) that updates the state in the session. Adds the Goto Element (and Connections) that redirect the flow to the evaluate state step of the Composition Template.
Connection	Adds a Connection.
StartElement	Adds a <SkeletonStartElement> element.
ConditionalElement	Adds a <SkeletonConditionElement> element.
CompositionSessionCommandElement	Adds a <SkeletonSSMElement> element.
GotoElement	Adds a <SkeletonGotoElement> element.
EndElement	Adds a <SkeletonEndElement> element.
ServiceElement	Adds a <SkeletonServiceTemplateElement> element.
ServiceConstraints	Adds the content for a <ServiceConstraints> element. Current implementation supports single constraints only.
CallParameters	Adds the content for a <CallParameter> element. Not implemented. Added as interface for future enhancements.

Table 2, Overview Named Templates

When a Named Template is processed through the <call-template> instruction, parameters can be set. This is typically used to indicate whether a Named Template should process Composition Template Elements or Connections.

6.5.2.3 Composition Template Elements, Connections and Unique IDs

The Composition Template document contains Composition Template Elements (vertices) and Connections (edges). Composition Template Elements are identified by a unique ID. Connections describe a relation between a source Composition Template Element and a target Composition Template Element using this unique ID.

The unique ID for the Composition Template Elements is created through the use of a concatenation of one or more elements of the following list:

- the state name
- the trigger name
- the position of the current node in relation to its siblings (e.g. first child, second child, ...)
- a fixed string set in the XSLT transformation

XSLT is a declarative language. Global variables that can change value are not supported. (The variables and parameters used in the transformation are in effect constants that have a fixed value in the context in which they are specified.) The Composition Template document contains a number of Composition Template Elements and Connections. These are related through the unique IDs that identify a Composition Template Element.

Because the Composition Template Elements and the Connections are set in separate parts of the target document it is not possible to describe both a Composition Template Element and a Connection at the same time. As a result, the Composition Template Elements and the Connections are described in different parts of the XSLT transformation. However, the unique IDs used must still match.

To simplify design it was decided to use, to a large extend, the same Named Templates for both Composition Template Elements as Connections. Through the use of a parameter (with name `type`) a Named Template toggles between describing vertices or edges. Strictly taken, this is not necessary to enforce correct unique IDs, as these are generated based on the source document, but a single code base reduces the complexity and improves maintainability of the transformation.

6.5.3 Step 1, Evaluate State

The evaluate state step is performed by a Conditional Element with the fixed ID `sm_main` and condition `$(sm_state)`. The condition evaluates to the value of the stored state. The Conditional Element has outgoing Connections for each of the states in the state chart. The Case's of the outgoing Connections are the name of the states.

6.5.4 Step 2, Evaluate Trigger

Three types of triggers are supported.

- SIP Request (only Initial INVITE SIP Requests are supported)
- SIP Response
- WS Response

The evaluate trigger step is performed by two Conditional Elements. The first Conditional Element with ID `<STATE>_CheckEventType` and condition `$(request_type)` evaluates the request type which is a system parameter. The result is either `WS` (for Web Service) or `SIP` (for a SIP Request or SIP Response). For SIP Messages, the second Conditional Element with ID `<STATE>_IsSipResponse` and condition `=*$(sip_response)` evaluates the type of the SIP message. The condition evaluates to TRUE if a SIP Response is available and FALSE otherwise.

An End Element with ID `<STATE>_<TRIGGER>_ILLEGAL_STATE` is added for those trigger types that are not specified for the current state. This End Element is a placeholder for a more sophisticated error handling.

6.5.5 Step 3, Evaluate Guard

The evaluate guard step is performed by a Conditional Element with the fixed ID `<STATE>_<TRIGGER>`, condition `$(sip_request_method)` for triggers of type SIP Request and condition `$(ws_result)` for triggers of type WS Result. The condition evaluates to the value of the guard for the current state and current trigger. The Conditional Element has outgoing Connections for each of the guards of the current state and trigger in the state chart. The Case's of the outgoing Connections are the value of the guards.

There is a potential risk as the transformation does not automatically add a default branch to the Conditional Element. If the event has a SIP Request method or Web Service result that is not specified in the State Chart, then the execution flow of the Composition Template will remain at the Conditional Element. Any further behavior of the Composition Template would not be conform a state machine. It could be argued this is a problem with the state chart. If the state chart does not specify all possible guards, the state chart is invalid. The transformation could be hardened to check for this situation and add a default branch to an End Element or some error handling. For simplicity reasons this is not implemented.

The evaluate guard step is different for triggers of type SIP Response. The Conditional Element evaluates a condition, and matches the result with the values of the Case attribute of the outgoing Connections. The guards for a SIP Response trigger are typically ranges. For example all SIP Response status codes between 100 and 700. Although creating 700 outgoing Connections (one for each possible SIP Response status code) would generate a valid Composition Template, this could be considered inefficient. This is illustrated in Figure 34 (not all possible SIP Response status codes shown).

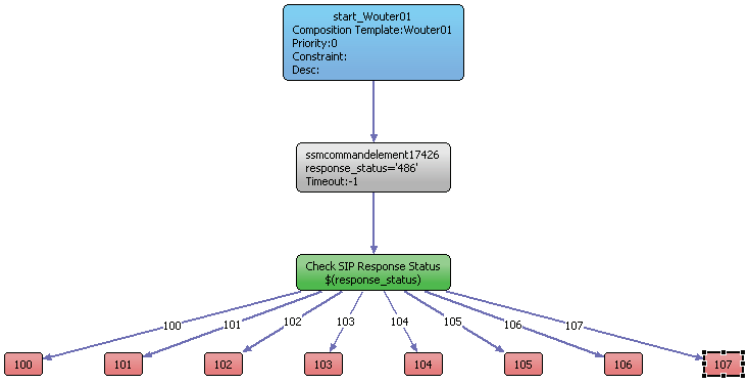


Figure 34, Basic Conditional Structure

The transformation uses a more efficient procedure to evaluate the guards for a trigger of type SIP Response though multiple Conditional Elements. Each guard creates a Conditional Element with the fixed ID `<STATE>_<TRIGGER>_<GUARDPOSITION>` and as condition the value of the guard. The condition evaluates to TRUE or FALSE. The TRUE branch connects to the effects for the given state, trigger and guard. The FALSE branch evaluates the next guard for the given state and trigger, or evaluates to an End Element with ID `<STATE>_<TRIGGER>_ILLEGAL_STATE` if no more guards are available. This is illustrated in Figure 35.

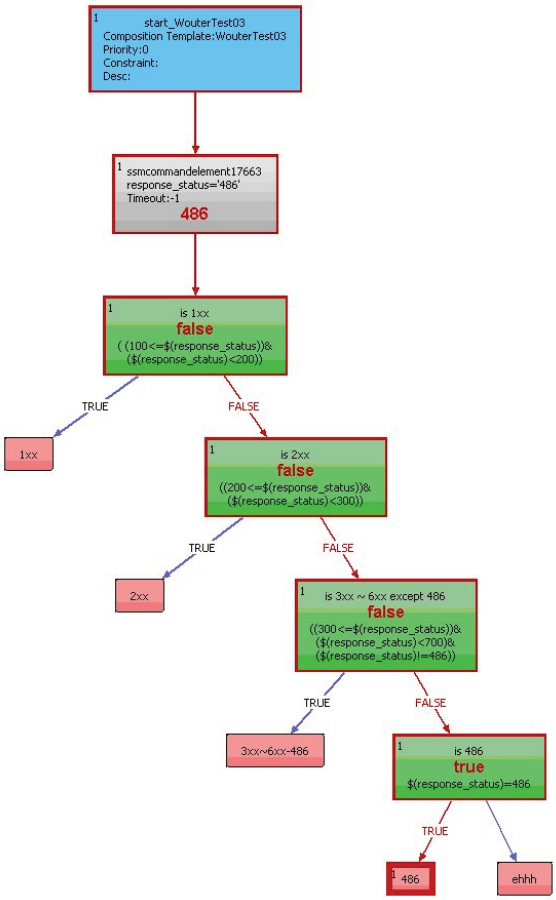


Figure 35, Hierarchical Conditional Structure

6.5.6 Step 4, Execute Effects

The execute effects step is performed by adding a Service Element (and Connections) for each effect. The ID of the Service Element is `<STATE>_<TRIGGER>_<GUARDPOSITION>_<EFFECTPOSITION>_EFFECT` and the constraint of the service element is given the value of the effect. No Service Elements are added if no effects are specified.

A Composition Template allows multiple constraints to be set in a Service Element. The transformation only uses a single constraint.

The SCE DSL does not include a "wait for next event" feature. Unless the execution flow of a Composition Template blocks until the next event is received, there is a risk of looping. That is, an already processed event is processed again. This result in undefined behavior of the state machine and may cause an infinite loop.

A "wait for next event" feature, or more precise a feature that blocks execution of the Composition Template until the next event is received, can be implemented through the use of special effects. Such an effect is the "Leave Container" effect which causes the SIP Message (either SIP Request or SIP Response message) to leave the SCE Engine. The execution flow of the Composition Template will block until the next event is received. Depending on the type of trigger, different mechanisms are needed.

For the effects of a trigger of type SIP Request, it is the responsibility of the State Chart specification to include the "Leave Container" effect. The "Leave Container" effect is closely related to the SIP functionality to setup a call and is therefore an integral part of the Telecommunication Service specified by the State Chart.

The effects of a trigger of type SIP Response are similar to the effects of a SIP Request and for the same reasoning it is up to the State Chart to include the "Leave Container" effect.

For the effects of a trigger of type WS Response no "Leave Container" effect is required. The execution of a Web Service is synchronous and the execution flow of the Composition Template blocks until the Web Service returns. An event with trigger WS Response is created when the Web Service returns.

An exception is a trigger for which a guard has no effects. For example, a provisional SIP Response would typically "do nothing". Having to add "Leave Container" effect would be counter intuitive. Therefore adding the "Leave Container" effect for this situation is a responsibility of the transformation. For triggers and guards that have no effects specified, the transformation will add a Service Element with ID `<STATE>_<TRIGGER>_<GUARDPOSITION>_EFFECT` and a constraint with value "LeaveContainer".

Adding a "Leave Container" effect for triggers and guards with no effects causes a problem. If the next state is a Final state, this state will only be reached if a subsequent event is received. A more advanced mechanism would be to: Add a "Leave Container" effect for triggers and guards with no effects and for which the next state is not a Final state. For simplicity reasons, the more advanced mechanism is not implemented in the transformation.

6.5.7 Step 5, Update State

The update state step is performed by adding a Composition Session Command Element with ID `<STATE>_<TRIGGER>_<GUARDPOSITION>_UPDATE_STATE`. This element stores the next state in the session.

A Goto Element is also added to continue the execution flow with the check state step, that is, the Conditional Element with ID `sm_main`.

6.5.8 Call Forwarding On Busy Service

As an example the Call Forwarding on Busy Service is designed as a UML State Chart Model. The State Chart diagram is shown in Figure 36. Note that not all effects or state names are visible.

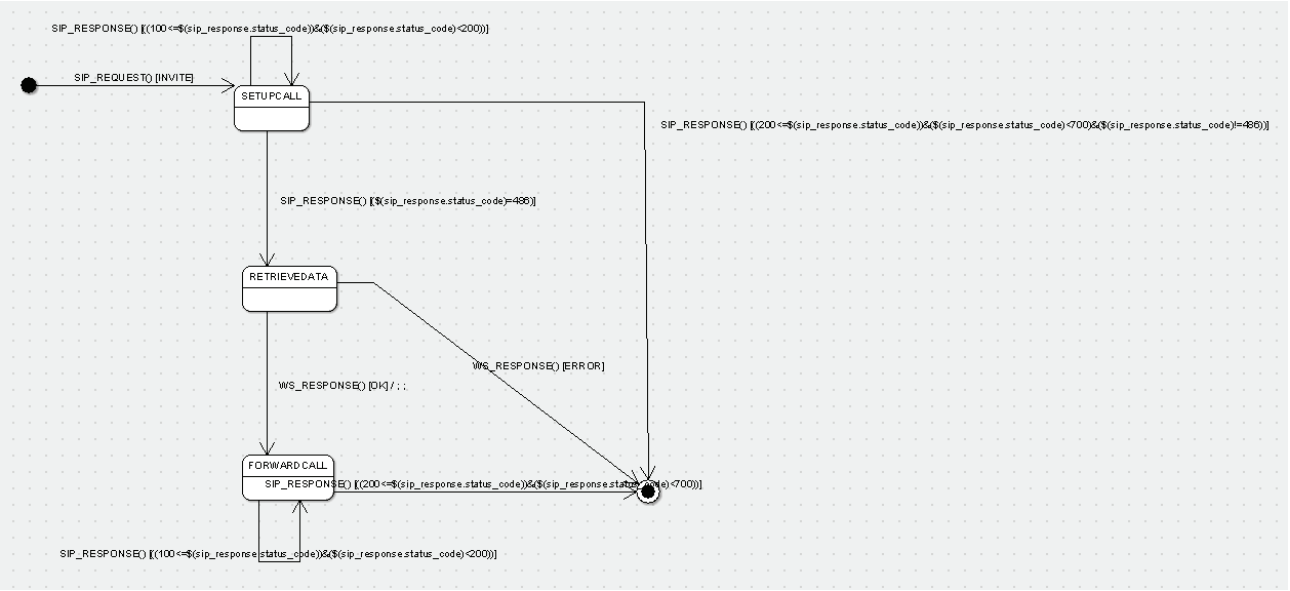


Figure 36, State Chart Diagram CFB Service

After transformation, the Composition Template document can be imported in the SCE Design Environment. A visual representation of the Composition Template representing the Call Forwarding on Busy Service is shown in Figure 37. The structure is basically the same as the flow diagram shown in Figure 33.

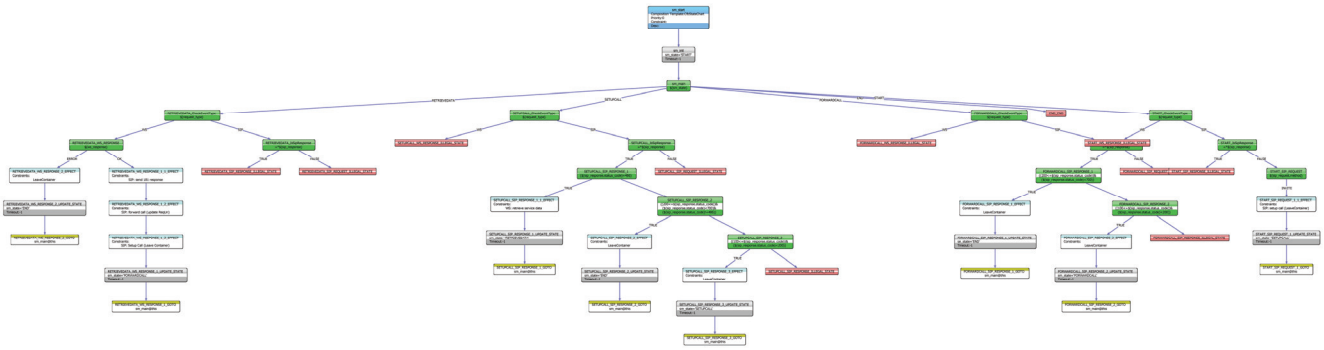


Figure 37, Visual Representation Composition Template CFB Service as generated through XSLT

7 SCE State Machine Transformation Overview

7.1 Introduction

This chapter describes the approach, design choices and observations made during the analysis and design of the transformation of a UML State Chart into a Composition Template implementation.

First the initial observations and approach are discussed, followed by more detailed observations made during the analysis and design.

A description of the SCE Transformation itself is provided in chapter 6.

7.2 Initial Observation

A state chart or state machine is based on states, transitions, triggers, guards and effects. The SCE DSL supports a *session* (to store the current state), a C*onditional element* (to choose a different branch depending on the state or transition), a *Service Template element* (to obtain an effect) and a *Goto element* (to continue processing at the beginning of the state machine implementation).

With these components it was expected a state machine framework can be implemented as a Composition Template.

7.3 Approach, Input, Output and Tools

This chapter describes the approach, general design choices and the environment through which the SIP Service is transformed.

7.3.1 Approach

The example Call Forwarding on Busy service was already examined (see chapter 5). To confirm that this service could actually be implemented as a Composition Template, a reference implementation was developed.

The model was based on the standard UML State Chart model combined with insights gained during development of the reference implementation. This resulted in a number of model design guidelines.

Two efforts were made for the transformation of the model into a Composition Template. The first transformation was based on a Java application. The second transformation was based on XSLT.

The following steps were identified to investigate the SCE transformation.

1 Create a reference Composition Template implementation.

2 Create a state chart to be used as input of the transformation.

3 Create java transformation implementation (imperative).

4 Create a more formal XSLT specification (declarative).

5 Analyze and test of the generated Composition Template.

These steps are further discussed below.

7.3.1.1 Create a reference Composition Template implementation.

The purpose of a reference implementation is twofold. First, it confirms the (example) Call Forwarding on Busy service can be implemented using SCE. Secondly, it forms a baseline for the development of the transformation. The insights gained during development of the reference implementation also assisted in design of the State Chart Model.

Two reference implementations were designed. The first implementation used a simple structure and was intended to confirm the service could be implemented as a Composition Template. The second implementation had the same behavior, but was structured as a State Machine. This design was used as a basis for the transformation.

A Composition Template contains Service Elements that reference Constituent Services. These runtime executable objects perform the actual *action*s of the model. A set of Constituent Services that performed basic SIP functions such as "proxy", "send response" or "forward" were created and used both by the reference and the transformed Composition Templates.

One of the main questions is *how* to implement a State Machine as a Composition Template. Several alternatives are available. It could be implemented using existing Composition Template features. The State Machine could be implemented as a separate SIP Application that is included in a Composition Template using a regular Service Template Element. Or anything in between, for example a hybrid between a specialized SIP Application or new Composition Template feature and regular Composition Template features.

From a practical point of view it was decided to use atomic Constituent Services that perform a single task. The logic of the service is fully contained in the Composition Template.

Another design issue was the choice between a single Composition Template and multiple Composition Templates. As the Call Forwarding on Busy service is part of the set of Call Diversion services it was considered to use multiple Composition Templates. A "master" template would decide which service should be executed and in which order depending on some kind of subscriber profile. Sub templates would then be used to perform the various services.

In the end it was decided to design a single Composition Template that operates as a stand alone CFB service as the purpose was to create a service implementation as a proof of concept and a full blown composition framework would not serve that purpose.

7.3.1.2 Create a state chart to be used as input for the transformation

The OMG UML specification was used as format for the state chart and the OMG XMI specification was used as format to store the state chart. A UML State Chart supports all the required functionality of state charts. UML also matches standards used in the customer environment.

A number of variations are possible for representing the Call Forwarding on Busy service as a state chart. A compact form was chosen to base the transformation on. This state chart was updated several times due to new insights during analysis and creation of the transformation.

7.3.1.3 Create java transformation implementation (imperative)

The transformation takes the state chart represented in a XML file as input and transforms this into an output XML file that represents a Composition Template. The input file conforms to the XMI specification and the output file conforms to the XML Schema file for SCE Composition Templates.

A Composition Template that represents a state machine was created manually using the SCE design environment. These first versions focused on the general flow and were not fully functional. Even small state charts with just states already lead to Composition Templates with numerous elements. It is not feasible to create these Composition Templates manually.

A Java application was created to automate this process and, basically, this was the first functional transformation tool. In its first draft a basic "proof of concept" transformation that focused on the general flow was supported. In later drafts of the Java application, more enhanced features were added as a work-around for certain problems.

7.3.1.4 Create a more formal XSLT specification (declarative)

The correctness of the transformation is not proven. It is corroborated by testing and by providing the specification of the transformation.

Due to the structure of the Java application, the source code is not appropriate as a specification of the transformation. The focus while creating the Java application was on a functional tool and not on creating a descriptive structure.

Several techniques exist to specify the transformation. Besides an imperative specification (e.g. the source code of the Java application), a declarative specification was considered to be more useful. This could be a mathematical description, for example a formal language, with rules describing the transformation. Some analysis was done using the program transformation features of the CWI Meta Environment. Eventually, the XSLT standard for XML transformations was chosen to describe the transformation in. The XSLT allows for a clear declarative description of the transformation, XSLT specifically aims for transformation from and to XML files and XSLT is a well supported standard, for which implementations are available.

7.3.1.5 Analyze and Test

The generated XML file which represents a Composition Template must be well formed and its behavior during execution must match the behavior of the IMS Service represented by the model.

The generated XML file is well formed as it conforms to the XML Schema file and can be successfully parsed by the SCE.

The dynamic mapping of constraints still need to be set manually in the SCE design environment before the Composition Template is ready for execution, but this is by design.

The SCE design environment contains a debug mode that can be used to test a Composition Template. This debug mode allows specifying and simulating SIP Requests that are then executed by the Composition Template. Tests were started on the generated Composition Templates using this debug mode but could not be completed due to a SCE limitation (see chapter 7.3.2).

7.3.2 SCE Limitations

We found that in the SCE implementation the handling of SIP Responses is not operational. Implementation of this feature in the SCE was not expected within the timeframe of the assignment. Lack of this feature is blocking for testing of the "Call Forwarding on Busy" Composition Template as this service depends on the "busy" SIP Response.

Basically, the created state machine cannot be tested. A number of assumptions were made concerning, for example, the synchronous nature of service elements. These assumptions cannot be verified (or discarded).

In addition, the current SCE implementation does not support SIP Request events, except for the initial INVITE SIP Request. In particular, the BYE and CANCEL SIP Request methods are not supported.

The SIP protocol is based on a-synchronous messages. In addition, events such as time-outs may occur. As a state machine implementation is based on event handling, it is critical some kind of event handling mechanism is available.

The current implementation of SCE does not seem to support any event handling mechanism. A state machine implementation of a Composition Template, or any Composition Template for that matter, cannot handle subsequent events while still processing the first event. This cannot be solved on Composition Template level and is a limitation of the SCE itself.

7.3.3 Input

The following documents and references were used during the investigation.

- Detailed Student Assignment PA7 [2]
- ETSI / 3GPP / IETF specifications of IMS and SIP [10]
- ETSI / 3GPP / IETF specification of Multimedia Telephony Services. E.g. Call Forwarding on Busy service [3]
- W3C specifications. E.g. XML, XSLT, XPath [5] [6] [9]
- OMG specifications. E.g. UML, XMI [7] [8]
- Java Specification Requests JSR-116 and JSR-289. Sip Servlet API specification [12] [13]
- Documentation of the Service Composition Environment
- Input from discussions with Ericsson Supervisor, University Supervisor, Ericsson SCE team and colleagues.

7.3.4 Output

The following documents were created during the investigation or generated as a result of the transformation.

An overview of the artifacts created as part of the investigation is provided in Appendix A.

- [uml] Design of CFB service. (state charts, sequence diagrams, serialized as XMI document)
- [code] Implementation of CFB service. (SIP Application, Composition Template)
- [code] Implementation of CFB service as State Machine (Composition Template)
- [code] Composition Template XML Schema document
- [code] XSLT transformation documents

7.3.5 Tools

The following tools were used for the SCE transformation.

- The SCE Engine and Design Environment, drop CEE 001
- ArgoUML v 0.24, UML editor
- Orangevolt XSLT plug-in for Eclipse, XSLT editor and engine
- Eclipse, integrated design environment
- Sun Java 5.0
- Ericsson SDS 4.0, SIP Service design environment including IMS emulation and testing framework.
- BEA Weblogic 9, JSR-116 compliant application server

7.4 Observations

7.4.1 On the State Chart Model

For the input model the UML State Chart diagram is used. Only a subset of the UML feature set is used. Besides the structure of the State Chart itself, only some properties such as a state name, trigger type or action are set. More advanced use of the State Chart is not necessary to (ultimately) generate a Composition Template implementation.

Through the use of design guidelines (see chapter 6.3.1), a specific design of the state chart model is enforced. This simplifies the transformation and structure of the Composition Template state machine implementation. These design guidelines may have been chosen a bit too strict. Design of more complex State Chart models was limited by the inability to construct transitions without a trigger. A production level metamodel may benefit from a broad support of the UML State Chart features.

The reason to define the design guidelines (or constraints on the metamodel) is to reduce the complexity of the transformation. Two categories of constraints can be identified:

- Constraints that reduce the complexity of the model.

Only a subset the UML State Chart metamodel is allowed. This reduces the complexity of the model and subsequently reduces the complexity of the transformation that uses this model as input. For example, *actions* are only allowed for *transitions* and not for *states*.

- Constraints that reduce the complexity of the transformation.

  The *guard* property of the IMS Model is defined in the syntax of the target language. This removes the need for a transformation from an implementation neutral pseudo language into the target language and reduces the complexity of the transformation.

7.4.2 On the XSLT Transformation

The transformation from the UML State Chart Model to a XML Composition Template document is based on XSLT.

The UML State Chart is serialized as a XMI document. Although XMI documents may depend on the UML editor from which the document was exported, this was not a problem for the analysis as only a single editor was used. However, it should be taken into account that switching UML editor or UML version may require an update of the XSLT transformation.

The XMI document was found to be complex due to the need to correlate data elements in separate trees inside the document. The XPath query language is powerful enough to extract the required data. However, due to the number of XSLT elements needed and the length of the XPath queries, the resulting XSLT document becomes complex and difficult to maintain.

It was decided to split the transformation into two steps.

The first step transforms the XMI document into an Intermediate XML document. This Intermediate XML document contains only the required model information in a simple hierarchical structure. The XSLT transformation is limited to querying the required data from the XMI document.

The second step transforms the Intermediate XML document into the XML Composition Template document. The XSLT transformation is limited to constructing the XML Composition Template. Due to the simple structure of the Intermediate XML document, the XPath queries do not add to the complexity of the XSLT document.

7.4.3 On the Service Composition Environment

7.4.3.1 SCE Engine

While the Composition Templates seems simple with just a few different elements combined in a flow diagram, and, indeed, the visual language described in chapter 4 is structured and concise, the exact behavior of the Composition Template is dependent on the implementation of the SCE Engine and a list of exceptions and special keywords. The exact behavior of SCE is unclear for certain use cases.

7.4.3.2 "Leave Container" Service Element

The flow of the Composition Template state machine structure depends on the ability to "wait" for a next event. Such a function was assumed to be available through the use of a Service Element with a special keyword: "Leave Container". It was found that, depending on the last performed action, different constructions are needed to implement a "wait" for next event function.

- If the last action was a Web Service Request, no "Leave Container" Service Element is needed. The Web Service Request itself is blocking.
- If no action is performed, the "Leave Container" Service Element is required to interrupt the Composition Template flow.
- If a SIP action was performed, no "Leave Container" Service Element is needed as the SIP action already implies an interruption of the Composition Template flow.

The input model does not need to include "Leave Container" Service Elements. If required, this element is automatically added during the transformation.

Note: It is assumed the "Leave Container" feature can be used to "wait" for a next event as this could not be tested.

7.4.3.3 Final State

If the state machine has reached the final state, this does not mean the call session is closed. The final state indicates the Composition Template does not perform any action. Due to the nature of the SIP Chain, the call session continues until the session is closed. From a state machine perspective, the "end" state is not a UML *final state*, but a *simple state* that has only one outgoing transition that loops back to itself.

7.5 Conclusion and Remarks

We have constructed an IMS metamodel based on the UML State Chart metamodel simplified through the use of constraints that we call Design Guidelines. These Design Guidelines are a design choice with the aim to reduce the complexity of the model and subsequently reduce the complexity of the transformation that uses the model as input.

The transformation is split in two steps to separate data retrieval and state machine construction. This modularization of the transformation also limits the impact of changed format of input documents or changes to the target language.

We have created a transformation which generates a Composition Template that corresponds with the reference Composition Template. The correct behavior of both the reference Composition Template and the generated Composition Template could not be verified through testing.

The Service Composition Environment is still in development and as such not all features are implemented. In addition, a number of features required to execute a state machine based Composition Template in a production environment is missing. In particular asynchronous event handling and exception handling are missing.

8 Repleo State Machine Transformation Specification

8.1 Introduction

This chapter provides a description of the transformation from a UML State Chart to a Java state machine implementation using Repleo generation.

An overview of the observations and design choices made is provided in chapter 79.

The transformation is similar to the SCE State Machine transformation as described in chapter 6. The main differences are listed below:

- The transformation is based on a Repleo engine instead of XSLT.
- The transformation results in Java source code instead of a SCE Composition Template.
- Some minor changes were made to the UML State Chart model.

This chapter starts with an introduction of the new technologies involved, continues with a discussion on the changes to the UML state chart, and finishes with a detailed description of the transformation from model to source code.

The term *Repleo Transformation* is used to describe the transformation from UML State Chart (represented as a serialized XMI document) into generated Java source code. The term *Repleo Generation* is used to describe the generation of Java Source Code by the Repleo engine based on an ATerms input data file. The Repleo Generation is a (single) step of the Repleo Transformation.

8.2 Repleo

Repleo is a syntax safe template based generation system [22]. The generation of source code is based on a template with placeholders that references elements from a model. The implementation consists of a parser that derives a syntax tree from a combination of an object language grammar (such as the Java grammar) and a meta language, and an evaluator for processing the placeholders.

We use Repleo as a technology and tool to generate Java source code based on a UML Model and a set of templates. As such, we consider Repleo as a *black box*. An overview of the Repleo environment from a designer's point of view is given in Figure 38.

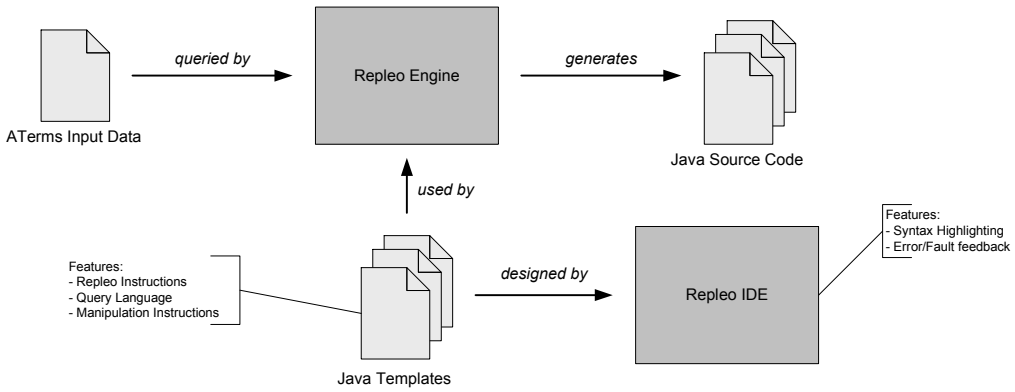


Figure 38, Overview Repleo Environment

The interfaces of Repleo, again from a designer's point of view, are listed below.

- The Repleo IDE

  The Repleo IDE is used to create template files through a design environment providing feedback such as syntax highlighting and visualization of (syntax) errors.

- The Repleo Engine.

  The Repleo Engine is used to generate source code based on templates and a model represented by the ATerms Input Data.

The template artifact consists of a mix of object language fragments and Repleo Meta Language placeholders. The Repleo Meta Language consists of:

- Repleo Instructions, e.g. Substitution, Conditional Selection, and Iteration
- Repleo Query Language, e.g. APath
- Repleo Manipulation Functions, e.g. _cc()

The Repleo Engine generates source code based on the templates and model. The model is represented as a data input file in the ATerms format [23]. ATerms is a hierarchical data structure for optimized data exchange between applications. Repleo uses a stricter variant of ATerms that consists of either lists or typed leaves.

8.3 Java State Machine

The target format of the transformation is Java source code which, after compilation, implements a state machine. The state machine is based on the State Pattern [16] from which the structure diagram is reproduced in Figure 39.

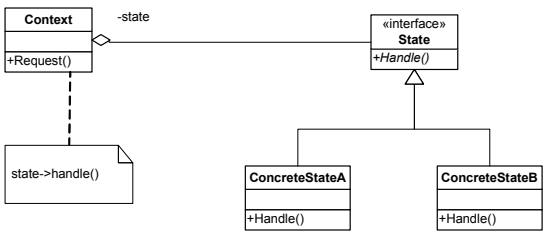


Figure 39, State Pattern Structure

While the standard State Pattern describes a possible structure that can be used in a stand alone application, some adaptations are needed for the pattern to be used in a SIP Service context.

Each state of the state machine is implemented as a separate class named ConcreteStateA, ConcreteStateB, and so on, in Figure 39. The concrete states implement a State interface. The Context class provides an external interface to the state machine and stores an instance of the current state. The Context class implementation does not know any behavior of the SIP Service, but serves as a facade to the Concrete State representing the current state.

The state machine implements a SIP Service and is developed as a SIP Application. A SIP Application is a Java Servlet that conforms to the SIP Container interface JSR-116 [12]. The Context class must conform to the interface provided by the JSR-116 specification. As the Context class is not a stand alone Java application but is deployed as a Servlet on a Java application server, it has to store the instance of the current state in a session provided by the SIP Container.

During a SIP session, events may be received (near) simultaneously. The State Pattern in itself does not provide synchronization and unless the SIP Application is specially designed to handle simultaneous events the behavior will become undefined.

The design of the SUN Java state machine implementation does not take synchronization issues into account. The purpose is to provide a concise prototype and not to design a production level application. By controlling the (test) environment in which the SIP Application is installed and tested it is guaranteed no synchronization problems occur.

8.4 UML State Chart

Similar to the SCE State Machine transformation, a UML state chart is the input for the Repleo transformation. The model is strongly based on the same metamodel used for the SCE transformation but with the following differences:

- Web Service events are not supported. While SCE provided explicit support for Web Service events, this is not available in the SIP Application. Web Services can still be used through the implementation of an *action*.
- The syntax of the guard for an event of type SIP Response has been changed to Java syntax in which the keyword `status` can be used.
- Guards are now optional.
- For a given state and trigger type, the guards no longer must cover the complete domain.

Except for the above differences, the same model design guidelines as for the SCE State Machine transformation apply (see chapter 6.3.1).

The UML State Chart is serialized as a XMI document which forms the input for the Repleo transformation.

8.5 Three Step Transformation

The transformation from UML State Chart model to Java source code is a three step transformation. The UML State Chart model is represented as a XMI document.

- The first step transforms the XMI document into a concise Intermediate XML document.
- The second step transforms the Intermediate XML document into an ATerms document.
- The third step is the transformation from the ATerms document into Java source code.

The transformation is visualized in Figure 40.

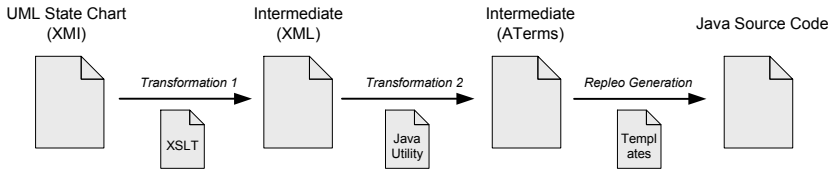


Figure 40, Three Step Transformation

Note that the Java source code generated in step three is not ready for deployment. The generated Java source code is syntactically correct and implements all *states*, *transitions* and checks for *events*. However, the generated Java source code does not implement the *actions* defined in the model.

8.6 Step 1: XMI to Intermediate XML Transformation

As mentioned in the description of the SCE transformation, this XMI document is complex. Only a subset of the available data is required, and related information is located in different parts of the document.

The query features of Repleo do not fully support queries and conditions on multiple branches of the input data. A transformation into a concise XML Intermediate document is a required step in the Repleo Transformation.

The transformation from the XMI document into an Intermediate XML document uses XSLT and is similar to the first step of the SCE Transformation. Both the XSLT document and the Intermediate XML format can be reused with little adaptation.

8.7 Step 2: XML to ATerm transformation

Repleo uses the ATerms format for the data input document. The tree structure of the input document uses nested lists and typed leaves. An example is given in Example 1. This example describes a list named `alfabet`. This list contains two leaves named `letter` of type `String` and with values `a` and `b`.

```
alfabet([
  letter(str("a")),
  letter(str("b"))
])
```

Example 1

The transformation between a XML document and the Repleo ATerms format is performed using a Java application. This application supports both a (simple) transformation from an XML document to a generic ATerms document, as well as a transformation to a Repleo ATerms document. The Repleo ATerms document has the additional constraints of nested lists and typed leaves.

The XML to ATerms transformation supports XML elements, XML attributes and XML chardata. For the Repleo ATerms format, all data is considered to be of type string. Some XML constructions cannot be transformed into a Repleo ATerms document. For example, mixing chardata (*leave*) with elements (*lists*) is not possible in the Repleo ATerms format. It was confirmed the transformed Intermediate XML document conforms to these restrictions.

Table 3 provides an overview of the transformation of various XML constructions.

XML	ATerms	ATerms (Repleo Format)
`<a/>`	`a([])`	`a([])`
`<a></a>`	`a([])`	`a([])`
`<a>abc</a>`	`a(["abc"])`	`a(str("abc"))`

`<a>`
`  abc`
`  <b>def</b>`
`  ghi`
`</a>`
	`a([`
	`  "abc",`
	`  b(["def"]),`
	`  "ghi"`
	`])`
		`N.A.`

`<a>`
`  <b>abc</b>`
`  <b>def</b>`
`</a>`
	`a([`
	`  b(["abc"]),`
	`  b(["def"])`
	`])`
		`a([`
		`  b(str("abc")),`
		`  b(str("def"))`
		`])`

`<a a1="abc" a2="def" />`	`a([`
	`  a1("abc"),`
	`  a2("def")`
	`])`
		`a([`
		`  a1(str("abc")),`
		`  a2(str("def"))`
		`])`

`<a a1="abc">`
`  def`
`</a>`
	`a([`
	`  a1("abc"),`
	`  "def"`
	`)]`
		`N.A.`

Table 3, XML To ATerms Transformation

8.8 Step 3: Repleo Generation

8.8.1 Introduction

The Repleo Generation takes as input a data file and one or more templates. The input data file is dynamic as it depends on the model. The templates that describe the structure of the generated source code are static. They are constructed for a specific usage. In this case the generation of source code that implements a state machine in a SIP Service context.

Because the templates depend on the structure of the implementation to be generated, this structure of the state machine implementation is described first. Next, the templates used for the Repleo generation are discussed.

8.8.2 State Machine Structure

The structure of state machine closely follows the State Pattern as described in chapter 8.3. The structure of the generated SIP State Machine is shown in Figure 41. The Context class (as seen in Figure 39) is implemented by the SipStateMachine class. As the SIP State Machine is a SIP Application deployed on a SIP Container, the SipStateMachine class extends the SipServlet class which is provided by the SIP Container.

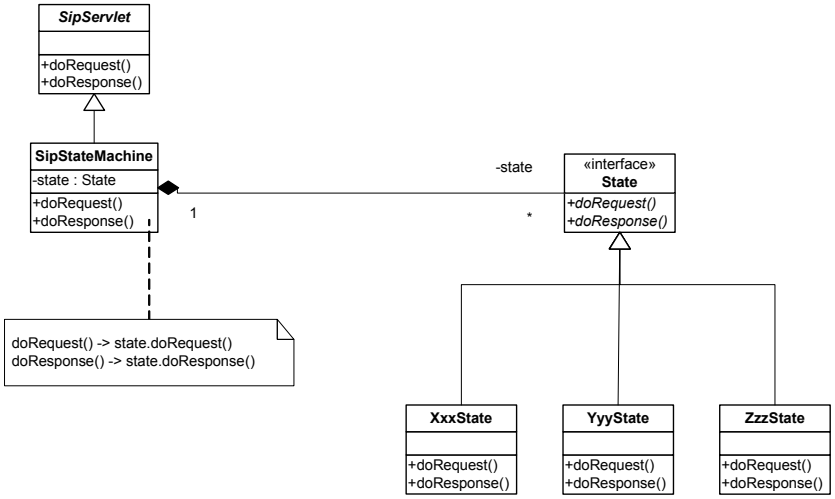


Figure 41, Generated SIP State Machine Class Diagram

The SIP Container, through the SipServlet class, provides an interface for the SIP events. The SipStateMachine class implements two of the provided methods: `doRequest` and `doResponse`. These methods cover the range of all possible SIP Request and SIP Response events. The implementation of these methods is a simple call to the corresponding method on the implementation of the current state.

As a design choice, the State interface defines similar methods: `doRequest` and `doResponse`. The semantics of these methods correspond with those provided by the SipServlet class.

For each state in the State Diagram a concrete state class is designed. These concrete state classes contain the behavioral implementation of their corresponding state in the model. These classes implement the `doRequest` and `doResponse` methods to handle SIP Request and SIP Response events.

The state chart may contain multiple transitions with the same trigger type for a single state. A one on one mapping of a transition with (for example) trigger type SIP_RESPONSE to a `doResponse` method is not possible. Instead the implementation of the method contains multiple code blocks. The correct execution flow is ensured through the use of conditional selection statements that check on the trigger guard. The design guide lines for the UML State Chart model prevent overlapping conditions.

The actions described in the model are in a free text format and cannot be transformed automatically into source code. Instead, the actions are added, in logical places, as comments.

For example, the following Intermediate XML fragment that describes a transition for a SIP_RESPONSE trigger for a "busy" status (status code equals the value 486):

```
<transition>
      <effects>
            <effect>Send "181 Call Being Forwarded" SIP Response upstream.</effect>
            <effect>Retrieve Forward URI and forward session.</effect>
      </effects>
      <trigger>SIP_RESPONSE</trigger>
      <guard>status == 486</guard>
      <target>FORWARDCALL</target>
</transition>
```

Example 2

will result in the following Java source code fragment:

```
if (status == 486)
{
      // ACTIONS

      // TODO: implement action
      // Send "181 Call Being Forwarded" SIP Response upstream.

      // TODO: implement action
      // Retrieve Forward URI and forward session.

      // UPDATE STATE
      sipStateMachine.setState(
            sipServletResponse, sipStateMachine.getFORWARDCALLState);
      return;
}
```

Example 3

A designer has to manually edit the generated Java source code and replace the markers with an actual implementation of the described action before the SIP Service implementation can be deployed.

8.8.3 Repleo Template Structure

The Repleo Generation results in Java source code that corresponds to the design described in the State Machine Structure chapter.

The Repleo Generation is based on three template files. These files and their purpose are described in Table 4.

Template	Purpose
`State_template.java`	Generates the `State` interface.
`SipStateMachine_template.java`	Generates the `SipStateMachine` class.
`ConcreteState_template.java`	Generates multiple `<statename>State` concrete classes.

Table 4, Overview Repleo Template Files

The SipStateMachine template only contains a minimum of generated logic. The class mostly keeps a list of instantiated concrete states and this list is created through Repleo generation. A simple foreach statement inserts the correct state names and the corresponding getter methods. The SipStateMachine template initializes the state variable to an initial state. The conditional foreach statement is used to query the input file for the correct state name.

The following example demonstrates the use of a foreach statement to initialize the concrete state variables. Note the use of the *lower case* string manipulation function to comply with Java naming guidelines. The replace statement is used both for a variable name as a class reference.

```
/*
 * initialize all states
 */
<%foreach data/states/state do%>
      <% _lc(name) || "State" %> = new <% name || "State" %>(this);
<%od%>
```

Example 4

The following example demonstrates the use of a conditional foreach statement to select the initial state from the data input file.

```
/*
 * Initialize state to start state.
 */
<% foreach data/states/state when type == "start" do %>
      state = <% _lc(name) || "State" %>;
<% od %>
```

Example 5

A template does not need to contain Repleo statements. The State template is almost Repleo statement free. Only a replace statement is used in the class javadoc section to insert the model name.

The Concrete State template allows generation of multiple files. A top level Repleo statement as well as statements in the filename section of the template file is possible. The following example is taken from the Concrete State template and causes Repleo to parse the template once for each state in the input data file. The state name is used to construct unique class names.

```
<% foreach data/states/state do %>
      com/ericsson/sipstatemachine/<% name || "State.java" %>
      ,
            ...
            Java source code / Repleo statements here.
            ...
<% od %>
```

Example 6

The Concrete State template generates the actual implementation of the service described through the state chart. All other classes are only implementing the State Pattern framework.

The model of the SIP Service, especially when seen in the simplified Intermediate XML document, is strongly hierarchical. This tree structure also forms the base of the Concrete State template. On the top level, iteration over the states creates multiple class files. Going down there are two iterations over the transitions. One for transitions of type SIP_REQUEST, the other for type SIP_RESPONSE. These implement the doRequest and doResponse methods. For a given transition, the generation finally iterates over the effects to generate the source code for all actions. Conditional Selection instructions are used to fine tune the generation.

Next to the static source code needed to implement the State Pattern, the Concrete State template also provides source code to create and initialize parameters that are used in the generated code blocks. For example the parameters which are checked against when evaluation the guard of a transition.

Based on the data input file and the (Java based) templates, the Repleo generation creates a number of Java source files which is at least the number of template files.

9 Repleo State Machine Transformation Overview

9.1 Introduction

This chapter describes the approach, design choices and observations made during the analysis and design of the transformation of a UML State Chart into a Java implementation.

First the initial observations and approach are discussed, followed by more detailed observations made during the analysis and design.

A description of the Repleo Transformation itself is provided in chapter 8.

9.2 Initial Observation

While the transformation into a Java implementation seems similar to the transformation into a Service Template, there is a major difference in the abstraction level on which the SIP Service is implemented, as is explained in the next paragraphs.

The Service Configuration Environment is a framework on top of the JSR-116 SIP Container. The framework defines its own DSL and provides an IDE to design Service Templates. A special SCE Engine is required to execute these Service Templates. The framework adds an abstraction layer on top of the container that simplifies service design, at the cost of hiding the *low-level* functionality of the JSR-116 SIP Container.

The Repleo transformation creates Java source code that is intended to be deployed as s SIP Application directly on the JSR-116 SIP Container. The SIP Container itself is deployed on a Java Enterprise Edition application server. Designing the service as a SIP Application allows use of the features of the SIP Container, but at the cost of the specialized features of the Service Configuration Environment.

The questions whether the design of such a SIP Application would be sound and concise, and whether the Java source code for said application can be generated through use of Repleo were still open at the start of the investigation.

9.3 Approach, Input, Output and Tools

This chapter describes the approach, general design choices and the environment through which the SIP Service is transformed.

9.3.1 Approach

The following steps were identified to investigate the Repleo transformation:

1 Investigate whether the state chart, used as input for the transformation, needs to be changed.

2 Create a Java reference SIP Application.

3 Create Repleo templates to perform the transformation.

4 Analyze and test.

9.3.1.1 Investigate the State Chart

The SCE contains features to handle Web Services requests and responses. While Web Services are supported by the SIP Container it was decided to limit the scope of the SIP Service implementation by not including Web Services support. The example state chart as used for the SCE transformation was updated by removing the *state* and *transitions* related to Web Services.

Further, the target language is Java. To simplify generation of Java source code it was decided to describe the guard for transitions of type `SIP_RESPONSE` using Java syntax. The guard can then be taken "as is" and placed in a placeholder without any additional processing. The guard could have been described using an implementation neutral syntax at the cost of added complexity to the transformation.

Concerning the actions specified for a transition, the SCE transformation contained actions described in a free text format. A designer needed to import the generated Service Template into the SCE design environment and set the correct constraints for the generated Service Elements.

A similar approach is used for the Repleo transformation. It is assumed the model describes actions in a free text format. This free text description is placed as comments in the generated source code. A designer has to manually edit the generated source code and replace the comments with an actual implementation for the specified action.

9.3.1.2 Create a Java Reference SIP Application

A reference implementation of an example SIP Service was designed in Java using a state machine design.

The purpose of this reference implementation is:

- to confirm a SIP Service can be implemented as a Java SIP Application using a state machine design
- to have a reference implementation as a basis for the Repleo templates

While (commercial) JSR-116 state machine based frameworks are available (for example ECharts for SIP Servlets [17]) it was decided to use the standard State Pattern [16] as a simple and transparent design structure.

9.3.1.3 Create Repleo Templates

The Repleo Templates are created based on Java reference source code. While this was not a conscious design choice, the Repleo environment was not available until work on the reference implementation was finished. In retrospect, adapting a reference implementation into templates seems to be the most logical approach when given a choice.

While investigating and designing the Repleo templates, it was found the query abilities of Repleo were unable to extract the required information from the UML State Chart (serialized as an XMI document).

In addition, Repleo was found to be unable to parse XML document as it requires input data to be in the ATerms format.

Two pre-processing steps were introduces to overcome these limitations:

- A transformation of the XMI document into a concise (and simpler) XML Intermediate document.
- A transformation of a XML document into an ATerms document.

The transformation from a *complex* XMI document into a *simpler* XML document is similar to the first XSLT transformation steps as discussed in the SCE transformation. Note that while the simplification was not strictly needed for the SCE transformation, the simplification is required for the Repleo transformation.

A Java stand alone application was developed to transform a XML document into an ATerms document. Only the syntax of the document is changed and no information is added or removed.

9.3.1.4 Analyze and Test

Testing of the generated source code is performed through the:

- Repleo engine errors during Repleo generation
- compilation of the generated Java source code
- deploying and testing the generated SIP Service

Repleo is assumed to be stable and correct (which it appears to be). Errors during Repleo generation do happen, but are indications of problems in the template files and data input file. Once the data input and template design is stable, there should be no Repleo errors.

By compiling the generated Java source code any syntactical errors are found. Environmental errors, such as an external library not found are not considered here. Note that using Repleo for code generation implies that syntactical errors should not occur.

The generated Java source code can be deployed as a SIP Service on a JSR-116 compliant application server. For generated code "as is" the deployment itself should succeed. For generated source code, for which the actions have been implemented by a designer, an end-to-end test case is performed. The reference to the test case is available in Appendix A.

9.3.2 Input

The Repleo transformation continues from the SCE transformation investigation. For input documents, see also chapter 7.3.3. In addition:

- Repleo documentation [22]
- State Design Pattern [16]
- SDS 4.0 Documentation [18]

9.3.3 Output

The following documents were created during the investigation or generated as a result of the transformation.

A detailed overview of the artifacts created as part of the investigation is provided in Appendix A.

- [uml] Design of CFB service (UML State Chart, serialized as a XMI document)
- [code] XSLT document describing the transformation from a XMI document into an Intermediate XML document
- [code] Java source code that implements a stand alone application to transform a XML document into an ATerms document.
- [code] Repleo templates used by the Repleo engine as a placeholder document to generate Java source code
- [code] An XML document that contains test cases used in the SDS 4.0 IDE.
- [generated] The Intermediate XML document
- [generated] The Intermediate ATerms document
- [generated] The (generated) Java source code

9.3.4 Tools

The following tools were used for the Repleo transformation.

- ArgoUML v 0.24, UML editor
- Orangevolt XSLT plug-in for Eclipse, XSLT editor and engine
- Eclipse, integrated design environment
- Sun Java 5.0
- Ericsson SDS 4.0, SIP Service design environment including IMS emulation and testing framework.
- BEA Weblogic 9, JSR-116 compliant application server
- Repleo 0.2.3

9.4 Observations

9.4.1 On the Model

The SCE State Chart model is reused for the Repleo Transformation. Some changes were introduced due to the difference in the target language or the description of UML actions. This in relation to the implementation choices made in the model as discussed in chapter 7.4.1.

As with the SCE Transformation, the model design guidelines result in a simple structured model. The model ensures no overlapping transitions are possible. E.g. either the guard is empty or the guards for a specific trigger do not overlap. This results in a relatively simple template structure for code generation.

9.4.2 On the XML to ATerms Transformation

The Repleo generation produces source code based on templates and a model. For this model, or data input file, the ATerms format is supported. To setup an end-to-end transformation it is necessary to include a transformation from XML format to ATerms format as the input data is only available as XML document.

This XML to ATerms transformation was not expected to cause any difficulties. Both formats use a similar hierarchical structure and it was somewhat expected a conversion tool would already be available. However, a search through the ASF SDF [20] and Stratego [24] technologies did not provide useable results [26].

An effort was made to construct a transformation through ASF+SDF term rewriting. This effort was abandoned due to difficulty with an overlap between ATerms grammar and the ASF syntax.

A Java application was constructed instead to transform a XML document into an ATerms document. The design of the Java application was based on the ATerms Java library and the Java API for XML Parsing (JAXP) [25].

The Java application supports both a generic ATerms output format as the stricter Repleo ATerms format that is based on typed fields and lists only. Table 3 illustrates some of the differences.

Some restrictions apply.

- As a design choice, all fields are considered to be typed as String.
- An empty field is not possible in Repleo. This can only be constructed as an empty list.
- It is not possible to mix fields and lists in Repleo, while mixing *chardata* (e.g. text) and *elements* is allowed within elements in a XML document.

These restrictions must be taken into account when designing the input data format.

9.4.3 On the Java State Machine Implementation Design

As the State Machine implementation design serves as a proof of concept, the standard State Design Pattern as described by the Gang of Four was preferred over other more specialized designs [17].

A reference Java implementation was developed as a baseline for comparing future generated implementations. The reference implementation was also used as a basis for template construction.

The State interface, as shown in Figure 39, is based on the top level methods to handle SIP Requests and SIP Response events. The Context class is a simple wrapper between the `doRequest` and `doResponse` methods of the SIP Container interface and the corresponding methods of the State interface. All conditional checking of the guards and implementation of the actions are implemented in the Concrete States.

The State Machine implementation is not a stand alone application but is deployed on a Java Application Server on which a JSR-116 compliant SIP Container is available. This deployment on an Application Server impacts the State Design Pattern.

- The Context class, which will be deployed as a Servlet on a Java Application Server, cannot store instance variables. Instead, the current state must be stored in a session provided by the Application Server. This is a technical detail and does not impact the concept of the State pattern.

- The Context class is deployed as a Servlet on the Java Application Server. For a SIP Application, the Context class must extend the `SipServlet` class. This `SipServlet` class provides its own interface of methods. Conceptual, there are now two "interfaces" in the State Pattern. The State interface and the interface provided through the `SipServlet` class. While it would be conceptually nice to only have a single "interface", it was decided, for simplicity reasons, to retain the State interface.

Even though there are some remarks on the State Machine implementation, it does serve its purpose as a proof of concept implementation. The design of the State Machine is simple and structured.

A production level implementation would require a design in which issues such as thread safety, synchronization and efficiency must be considered.

9.4.4 On Repleo

9.4.4.1 Introduction

The observations and experiences of the Repleo investigations are grouped according to the following topics:

- The Repleo IDE
- The Templates and Template Design
- The APath Query Language
- The Repleo Engine
- The generated Source Code

The investigation of the Repleo transformation was performed on version 0.2.3 of Repleo. It should be noted that Repleo is being developed and observations made related to this specific version may not be valid for newer releases of Repleo.

9.4.4.2 The Repleo IDE

Modern development environments such as Eclipse [19] provide syntax checking and other visual feedback on typed in source code. A developer takes this as granted and the syntax checking provided by the Repleo IDE may, at first glance, not be considered to be anything special. And for a developer this is exactly what is required, a simple and (mostly) intuitive development environment.

The Repleo IDE is an extension of the ASF+SDF Meta Environment [20]. The Meta Environment is intended for language development, source code analysis and source code transformation. The main artifacts are the syntax definition (*module*) and term files on a module that can be reduced ("parsed"). The Meta Environment contains useful features for module development and term file reduction such as a visual syntax tree representation.

Repleo provides a complete module that provides a specialized Java syntax module that allows the use of the Repleo instructions. Modules for a number of other languages are included in the Repleo delivery as well. The Repleo IDE is typically used to create (or edit) term files. These are named template files in the Repleo context. While the Meta Environment supports creating and editing term files, there is not much support or automation for the designer, except for the inherent syntax checking. Especially when compared to a Java (or other language) based IDE such as Eclipse that provides features such as code completion and checking.

We recommend creating a reference implementation using a specialized target language based IDE, and use these source code files as input for template design.

An example of the Repleo IDE is shown in Figure 42.

The panels on the left side are related to the Repleo module that provides the specialized Java syntax and are not used during template design. The panel on the upper right side of the screen shows the template and provides editing functionality. Java keywords, Repleo keywords and errors are highlighted. The panel on the lower right, and in particular the Issues tab, shows any syntax errors in the active template. In the example a syntax error is triggered by a missing)-character. The description of the "parse error" contains detailed information such as the line and column of the error.

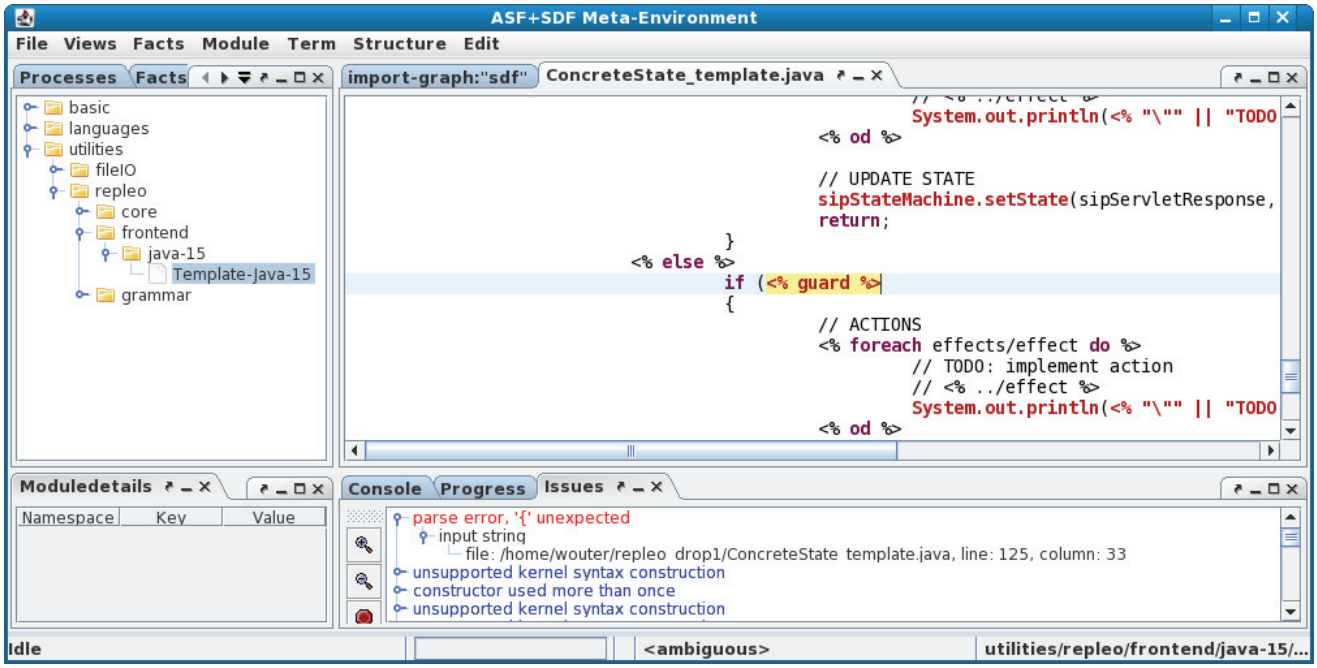


Figure 42, Repleo IDE (ASF SDF Meta Environment)

The Repleo IDE provides a functional, though plain, environment to edit template files. Syntax errors are immediately and clearly shown.

There are some usability remarks on the Repleo IDE.

- To design a template, the Repleo IDE / Meta Environment is started. Next a specific *Repleo module* must be loaded, and from this module the actual template files can be opened. From a Meta Environment perspective a logical approach, but otherwise somewhat less intuitive.

- The syntax highlighting could be improved. Currently the Java and Repleo instruction keywords are highlighted using the same color scheme. It would be useful if the Repleo instructions are clearly distinct from the rest of the source code.

- A number of warnings are shown in the Issues tab. Also, ambiguity errors are reported, and the corresponding source code in the term file is shown with a red font as text highlighting.

Most of these remarks are caused by the integration of Repleo on top of the Meta Environment. The purpose of the Meta Environment is to create modules (e.g. half of the visible panels in the environment are module specific), while the purpose of Repleo is to create templates. The use of Repleo instructions in the template files often causes ambiguities that are solved inside the black box that is Repleo, but the Meta Environment is still reporting these errors.

Solving these issues probably requires further customization of the Meta Environment, which may not be a priority.

9.4.4.3 The Templates and Template Design

The usefulness of syntax checking during template design is illustrated by the following experience.

As a preparation, a template file was created using a regular editor. Later, when the Repleo environment became available, this template file was loaded into the Repleo IDE and immediately a number of syntax errors were shown. This caused some surprise as the template files were carefully prepared and believed to be correct. After further inspection the indicated errors were found to be caused by some missing semi-colons. Repleo discovered and pinpointed an error during template design which otherwise would probably only have been found through some less specific errors during compilation, that is, during one of the last steps of the transformation process.

The design of the templates to generate the Java state machine implementation for a SIP Service was based on a reference implementation created outside of Repleo using the Eclipse based SDS 4.0 [18] [19] development tool. SDS 4.0 contains the Eclipse set of Java Language development features as well as IMS and SIP Service specific development, packaging and test features.

The reference source code was developed not only to provide a base to start template design on, but also to provide a baseline to compare the generated implementation with.

The structure of the Repleo generation is hierarchical or list based, in which a list could be seen as a shallow but broad tree, due to the format of the data input file. The structure of the reference implementation is by its state machine based nature hierarchical. During design of the reference implementation no changes were needed to the standard State Pattern design. The only real effort required to simplify the transition from the reference implementation to a template was not to optimize the source code as this would break the hierarchical structure.

Repleo drop 0.2.3 provides a small but sufficient instruction set that allows substitution, conditional selection, and iteration. In addition, there are some string manipulation functions.

The design methodology used to create the template files (from a reference implementation) is represented by the following steps:

1 Locate structural identical files and combine these in a single template file.

   The concrete states are a good example of structural identical files that can (and must!) be implemented as a single template file.

2 Within a template file, locate structural identical code blocks and replace these with a foreach instruction.

   Examples of structural identical code blocks that were replaced by a foreach instruction are the getter methods and state variables in the context class.

3 Locate the dynamic expressions and declarations and replace these with a substitution instruction.

   The substitution instructions were used in combination with a manipulation function to comply with the Java design rules in regard to the case of the first character of a class name or parameter name.

Creating the templates from the Java state machine reference implementation posed little difficulty. This was mainly due to the similar hierarchical structures of the state machine implementation and the model.

Repleo operates through the use of placeholders (e.g. Repleo instructions) and placement of these instructions defines the structure of the generated source code. The structure of the input data file must be fixed. For example Repleo cannot handle a dynamic depth of the input data

The template file with all Repleo instructions must be syntactical correct, even if the template would always generate correct Java source code. The fragment in Example 7 would cause an error. A similar construction

```
<% if guard == "" then %>
    if (true)
    {
<% else %>
    if (<% guard %>)
    {
<% fi %>
        // do something
    }
```

Example 7

The fragment could be rewritten as shown in Example 8 to be syntactical correct.

```
if (<% if guard == "" then %>true<% else %><% guard %><% fi %>)
    {
        // do something
    }
```

Example 8

The following list contains observations on Repleo instructions that were made during the Repleo investigation.

- Often, only a subset of items in a list was needed. This was implemented by using a foreach instruction followed by a condition selection instruction. A specialized `foreach-when` instruction was added by the Repleo designer to simplify this.

- The `<% else %>` part of the condition selection instruction was sometimes recognized by Repleo as the substitution instruction (`<% Expression %>`). It was recommended to change the syntax of the substitution instruction to, for example, `<% subst Expression %>`. A different solution was implemented by the Repleo designer. A list of keywords that cannot be used as a query was defined. It was observed that this could limit the input data file design as these keywords can no longer be used as elements.

- Two specialized statements were introduced (by the Repleo designer) to check on the existence of input data elements: the **`exists(`** `Query` **`)`** statement and the **`isEmpty(`** `Query` **`when`** `Expression` **`)`** statement. Parsing the input data file is further discussed in chapter 9.4.4.4.

- When using the foreach instruction, the place of the current element may be of importance. For example, the first element may need to generate an `if` statement, while the following elements should generate `else if` statements. Two new conditional selection instructions could be added. The first, for example `isFirst()`, would evaluate to true if the current element is the first element of the list, and the second, for example

`isLast()`, would evaluate to true if the current element is the last element of the list.

- It was found that during template design some source code fragments are used multiple times within a template. It could be useful to import fragments or to refer to a fragment inside a template.

- Some template files could be complicated. The use of Repleo comments could simplify the design. A comment instruction could be added, for example `<%% comment here %>`, that does not generate source code.

  Due to technical restrictions, the Repleo designer was unable to add comments as a Repleo feature. As a work-around, regular Java comments prefixed with the text REPLEO COMMENT are used instead in the example templates referenced in Appendix A. See also the comment in Example 9.

```
// REPLEO COMMENT: This is a Repleo comment.
```

Example 9

9.4.4.4 The APath Query Language

The APath Query Language (part of Repleo) is used from within Repleo instructions to navigate and query the input data file. The query language was found to be somewhat similar to a (simple version of) the XPath query language [6].

The input data file is a hierarchical structure of lists and typed leaves. For the Repleo transformation, and in particular as a result of the XML to ATerms transformation, the leaves are always of type string.

A query consists of an optional root indicator and one or more location steps. A location step is an ATerm list or leaf. Two specialized location steps indicate the current location, or the parent location. The query is parsed from the root list of the input data file if the root indicator is present, or from the current location otherwise. A query can indicate an ATerm list, leaf or is invalid.

For the Repleo transformation, the data input file resembles the hierarchical structure of a state chart model. Parsing was performed through nested foreach instructions. On the top level a foreach instruction looped through all states in the model. For a specific state, a foreach instruction looped through all transitions. For a specific transition, a foreach instruction looped through all actions.

The foreach instruction limits the scope to the current sub tree. Subsequent foreach instructions must be nested. Due to this constraint, it is not possible to parse the structure of XMI documents. In a XMI document, information must be retrieved by correlating elements from multiple (non-nested) sub trees.

The APath Query Language is used as an expression in the Repleo instructions. The instructions are typically used for two purposes:

1 To replace a placeholder with an element from the input data.

  The substitution instruction is used for this purpose.

2 To include or exclude blocks of source code in the template.

This is typically a combination of the foreach instructions to *navigate* to a certain location in the input data file, followed by one or more conditional selection instructions to *decide* on what code blocks should be included or excluded.

A SIP Service may react on triggers from the network such as SIP Requests or SIP Responses, but often the service performs does nothing, and just returns the trigger to the SIP Container which then performs a default action.

This translates in template design to a default code block that should be included if no actions are specified in the model.

The `exists` statement was added by the Repleo designer to check if input date (e.g. actions) exists. Later, a more specialized exists statement was added. The `isEmpty` statement operates over a list and allows an optional conditional expression.

```
<%% loop over all states %>
<% foreach states/state do %>

    <%% enable code block for transitions with a specialized trigger %>
    <% foreach transitions/transition do %>
        <% if trigger == "doErrorResponse" then %>
            // some java code block handling the specific error response
        <% fi %>
    <% od %>

    <%% enable code block for transitions with a generic trigger %>
    <% foreach transitions/transition do %>
        <% if trigger == "doResponse" then %>
            // some java code block handling the generic response
        <% fi %>
    <% od %>

    <%% make sure a default code block is enabled if no generic triggers %>
    <%% are available in the model %>
    <% if isEmpty( transitions/transition when trigger == "doResponse" ) then %>
        // a default java code block handling the generic response
    <% fi %>

<% od %>
```

Example 10

The `isEmpty` statement was requested for the following (simplified) use case. For a given state, a number of transitions are specified. Each transition has a trigger. This trigger can be a specialized trigger such as an error response, or a more general trigger such as "any" response. The generated source code must always include a code block to handle a general trigger. This use case is shown in Example 10.

9.4.4.5 The generated Source Code

The generation of source code is performed by the Repleo Engine. The Repleo IDE and Repleo Engine are separate entities. The Repleo Engine is accessed as a command line application.

The command line application (one for each language) accepts parameters to specify the input and output locations, and set various options.

Errors during generation (for example due to an incompatible data input file) are reported on the console. It was found that for some error conditions, the generated source code still contained Repleo instructions. For a production environment, the generated source code must be "correct", or errors must be generated.

From a process or automatic generation street perspective, it would be nice if key/value pairs could be set as command line parameters that can be accessed from the substitution instruction inside templates. This feature could be used to insert fields such as generation date/time, name of engineer, name of the project for which the source code is generated, etc.

Generation of the source code is very fast. The SIP Service source code was generated within 0.2 seconds on an AMD Athlon 2400+ based personal computer.

9.5 Quality

The quality of the generated source code was measured during the project by means of testing. Using the Automatic Testing Framework provided by the Service Design Studio (SDS) 4.0 a testing script was created and executed both on the reference implementation and the generated source code. Both the reference implementation and the generated source code were first manually packaged and deployed on the simulated IMS environment provided by SDS.

All test cases were passed for both the reference implementation and the generated source code. It was concluded that the behavior is in line with the expectations.

9.6 Conclusion and Remarks

It was found to be relative easy to create a transformation based on Repleo. The core functionality of Repleo, a syntax safe template engine, works correctly and creating templates is intuitive.

Repleo should be seen as part of a larger process that also involves pre and post-processing.

Pre-processing is required due to limitations of the meta language, and in particular the inability to execute complex queries. This is a design choice of Repleo to reduce the complexity of templates.

Post-processing is required due to limitations of the model. Post-processing involves completing those parts of the generated source code that could not be generated automatically.

The design of the generated source code is based on the classic State pattern. This is sufficient as a testable proof of concept. A production level implementation would require a more advanced design and thus more complex templates. No obvious obstacles were identified in regard to the Repleo instruction set or template construction that would prevent such a design.

Template design was performed using a complete reference implementation as base. The Repleo IDE is constructed on top of the ASF+SDF environment and as such is not primarily intended as a language IDE. Syntax errors were clearly shown. The limited set of Repleo instructions was sufficient to construct reasonably complex templates. Initially, some limitations were found in regard to conditional checking for non existing fields. This feature was included in newer Repleo releases.

The Repleo Generation suffers, seen from a developer's perspective, from a number of limitations. Java comments are lost and the generated Java source code misses whitespace information. While the missing whitespace information could be resolved by including a Java pretty printer as a post process, the dropping of Java comments is more critical. Java comments may contain important (post processing) instructions or could contain copyright information that must be present for legal reasons.

Testing for syntax errors was not performed exhaustively. Testing for typing errors was limited as the input data file was considered to only contain fields typed as String. It is assumed all syntax errors made during template design were identified by the Repleo IDE as "error free" templates consistently generated source code that compiled without errors when using a correct data input file.

10 SIP Semantics

10.1 Introduction

The purpose of the IMS Service Model is to describe an IMS Service. While this IMS Service is based on SIP, we observed that the amount of SIP related semantic information in the model is limited.

This was most obvious for the specification of the *action* (the effect), *event* (the trigger) and the *guard* for a *transition* between *states*. (These terms are defined by the UML State Machine formalism [7].)

For example, SIP related *actions* cannot be specified in a structured way in the model. And *events* are only identified through the use of arbitrary keywords.

This chapter investigates if and how SIP semantic information can be added to the model.

10.2 Background

The State Chart model as used to describe SIP services is designed to use keywords to indicate the event. While these keywords are clearly specified in the design guidelines as stated in chapter 6.3, they are introduced as an arbitrary choice. An action is described as a descriptive text, which may make sense for a human designer but is useless for any automated task.

It was believed that by adding SIP Semantics to the model, a number of improvements could be realized:

- The instance of the State Chart model as designed by the architect would be more in line with the SIP context.
- The design of the templates used for generation could become more structured and more advanced.
- The behavior of the service described in the model can be validated against the SIP specification.

Note that validation is outside the context of this thesis and is not further analyzed.

Two areas were identified to add additional semantic information to the model.

- By extending the base classes for actions and events with a number of stereotypes (see [7]).
- By adding an interface corresponding with SIP events and SIP actions. Actions and events can then map their operations to these interfaces.

The model was extended with both the stereotypes and an interface, although only for the events and for a reduced scope, as the purpose was for analysis and not to generate a complete production quality model.

Using this updated metamodel, the complete Repleo transformation was redesigned to analyze the impact of the added SIP semantics.

This chapter starts with a description of the SIP protocol and discusses the layered interface of SIP messages. Next, the updated State Chart model, and the impact for the architect is described. Finally, the impact on the state machine implementation structure and the Repleo templates is discussed.

10.3 Layered SIP Interface

The SIP protocol is based around SIP Messages. Other events are also part of SIP, such as time events or WEB Services, but these are not considered part of the scope of the analysis. SIP Messages fall in two categories: SIP Request Messages and SIP Response Messages. Both categories are further defined by properties such as the type of message and an initial or subsequent flag. Each received event is fully specified. For example an Initial SIP Request with method INVITE or a SIP Response with status BUSY. The SIP specification [11] defines a number of terms such as Final Responses that describe a group of SIP Messages.

The JSR-116 SIP Container [12] provides an interface towards SIP Applications. This interface, for SIP Responses, is shown in Table 5. A SIP Application may implement all, some or none of these methods. The SIP Container will trigger the most specific method first and continues with more generic methods until an implemented method is encountered. If no methods are implemented a default implementation is executed.

JSR-116 Interface Method	Description
`doProvisionalResponse`	For handling of 1xx SIP Response messages.
`doSuccessResponse`	For handling of 2xx SIP Response messages.
`doRedirectResponse`	For handling of 3xx SIP Response messages.
`doErrorResponse`	For handling of 4xx, 5xx and 6xx SIP Response messages.
`doResponse`	For handling of SIP Response messages.
`service`	For handling of SIP Response and SIP Request messages.

Table 5, JSR-116 Interface for SIP Responses (partial)

Both the SIP protocol as the SIP Container use a layered interface with a fall-through mechanism in which more specific methods are triggered first. In addition, the SIP Application does not have to provide any implementation for events. If no event is implemented a default implementation is executed.

10.4 Enhanced State Chart Model

For the SCE Transformation and the Repleo Transformation, the metamodel uses a rudimentary set of SIP Semantic information. SIP Requests and SIP Responses are recognized through the use of keywords (see Table 6). Further specification of the type of SIP Message is handled through the use of guards. These guards are defined by using some pseudo language or Java language which is easy to use in transformations, but does not really have any semantic value.

Keyword	SIP Semantic
`SIP_REQUEST`	The event is a SIP Request message.
`SIP_RESPONSE`	The event is a SIP Response message.

Table 6, SCE and Repleo Transformation - Keywords

The metamodel was enriched through the use of Stereo Types and introduction of an interface.

10.4.1 Stereotypes

Based on the SIP specification a number of stereotypes can be defined for the CallAction and CallEvent objects. The stereotypes indicate the type of Message (request or response), the type or a SIP Request and the status of a SIP Response. Further, stereotypes exist to indicate initial and subsequent messages.

Stereotypes can stack. For example, the stereotypes `<<SIP Request INVITE>>` and `<<SIP Initial Message>>` could be applied to a CallEvent to indicate an initial SIP Request message with method INVITE event occurred.

The list of stereotypes includes generalized events. Examples are the stereotypes `<<SIP Request>>` and `<<SIP Response>>`. The idea behind generalized stereotypes is mainly usability. To indicate a specific event is triggered by "any" received SIP Request, a single `<<SIP Request>>` could be applied, instead of having to apply all specialized stereotypes.

Stereotype with CallEvent as base class are listed in Table 7.

Stereotype	SIP Semantic
`<<SIP Request>>`	The event is a SIP Request message.
`<<SIP Request INVITE>>`	The event is a SIP INVITE Request message.
`<<SIP Request REGISTER>>`	The event is a SIP REGISTER Request message.
`<<SIP Request ACK>>`	The event is a SIP ACK Request message.
`<<SIP Request CANCEL>>`	The event is a SIP CANCEL Request message.
`<<SIP Request BYE>>`	The event is a SIP BYE Request message.
`<<SIP Request OPTIONS>>`	The event is a SIP OPTIONS Request message.
`<<SIP Request PRACK>>`	The event is a SIP PRACK Request message.
`<<SIP Request SUBSCRIBE>>`	The event is a SIP SUBSCRIBE Request message.
`<<SIP Request NOTIFY>>`	The event is a SIP NOTIFY Request message.
`<<SIP Request MESSAGE>>`	The event is a SIP MESSAGE Request message.
`<<SIP Request INFO>>`	The event is a SIP INFO Request message.
`<<SIP Response>>`	The event is a SIP Response message.
`<<SIP Response Provisional>>`	The event is a SIP Provisional Response message.
`<<SIP Response Successful>>`	The event is a SIP Successful Response message.
`<<SIP Response Redirection>>`	The event is a SIP Redirection Response message.
`<<SIP Response Request Failure>>`	The event is a SIP Request Failure Response message.
`<<SIP Response Server Failure>>`	The event is a SIP Server Failure Response message.
`<<SIP Response Global Failure>>`	The event is a SIP Global Failure Response message.
`<<SIP Response Final>>`	The event is a SIP Final Response message.
`<<SIP Response Error>>`	The event is a SIP Error Response message.
`<<SIP Initial Message>>`	The event is a SIP Initial message.
`<<SIP Subsequent Message>>`	The event is a SIP Subsequent message.
`<<Timer Timeout>>`	The event is a SIP Timeout message.

Table 7, Stereotypes for CallEvent

The stereotype `<<Timer Timeout>>` is included to demonstrate that stereotypes are not limited to message events but can be used for the full range of SIP events.

Stereotypes with CallAction as base class are listed in Table 8. These stereotypes represent the type of actions that can be taken on the SIP chain. These actions are similar to the events listed in Table 7, with the addition of Proxy actions and an indication whether an action involves the upstream or downstream call leg.

Stereotype	SIP Semantic
<<Send SIP Request>>	A SIP Request message is send.
<<Send SIP Request INVITE>>	A SIP INVITE Request message is send
<<Send SIP Request REGISTER>>	A SIP REGISTER Request message is send
<<Send SIP Request ACK>>	A SIP Request ACK message is send
<<Send SIP Request CANCEL>>	A SIP CANCEL Request message is send
<<Send SIP Request BYE>>	A SIP BYE Request message is send
<<Send SIP Request OPTIONS>>	A SIP OPTIOS Request message is send
<<Send SIP Request PRACK>>	A SIP PRACK Request message is send
<<Send SIP Request SUBSCRIBE>>	A SIP SUBSCRIBE Request message is send
<<Send SIP Request NOTIFY>>	A SIP NOTIFY Request message is send
<<Send SIP Request MESSAGE>>	A SIP MESSAGE Request message is send
<<Send SIP Request INFO>>	A SIP INFO Request message is send
<<Send SIP Response>>	A SIP Request message is send.
<<Send SIP Response Provisional>>	A SIP Request message is send.
<<Send SIP Response Successful>>	A SIP Request message is send.
<<Send SIP Response Redirection>>	A SIP Request message is send.
<<Send SIP Response Request Failure>>	A SIP Request message is send.
<<Send SIP Response Server Failure>>	A SIP Request message is send.
<<Send SIP Response Global Failure>>	A SIP Request message is send.
<<Send SIP Response Final>>	A SIP Request message is send.
<<Send SIP Response Error>>	A SIP Request message is send.
<<Send Upstream>>	The SIP Message is send upstream.
<<Send Downstream>>	The SIP Message is send downstream.
<<Proxy Proxy>>	A SIP message is proxied on the proxy.
<<Proxy Forward>>	A SIP message is forwarded on the proxy.
<<Proxy Cancel>>	The sessions associated with a proxy are cancelled.
<<Timer Set Timer>>	A SIP Timer is set.
<<SIP Do Nothing>>	The action is of type "Do Nothing".
<<SIP Not Implemented>>	The action is of type "Not Implemented".
<<Non SIP>>	The action does not involve SIP.

Table 8, Stereotypes for CallAction

It was observed that the use of Stereotypes allows describing the type of SIP events in more detail. Using Stereotypes has no effect on the structure of the model.

10.4.2 State Interface

A Class Diagram is added to the UML Model representing the SIP Service. This Class Diagram contains a specification of an <<interface>> *SipStateMachine* class (and supporting classes). While "SipStateMachine" is just an arbitrary name, the methods defined in the interface are carefully chosen to represent a range of SIP events. The Class Diagram is shown in Figure 43.

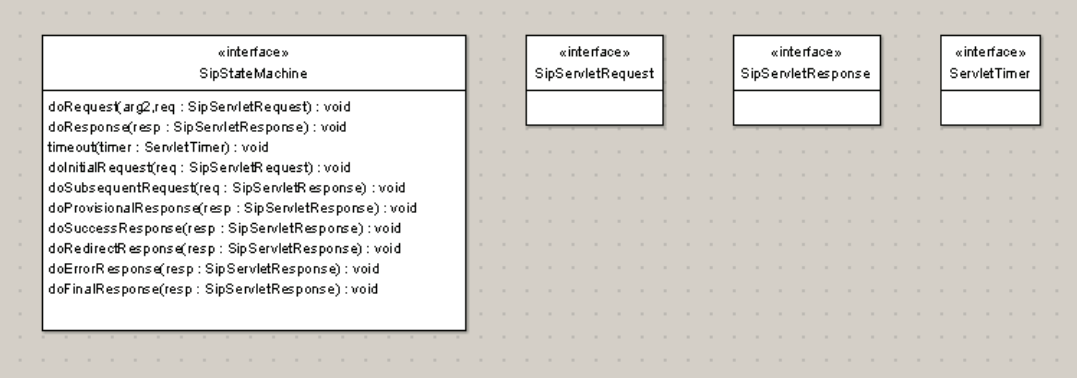


Figure 43, Class Diagram

The SIP semantics linked to the methods of the interface are shown in Table 9. By mapping each trigger used in the State Chart model to a method defined on the SipStateMachine interface the SIP semantic are added to the State Chart model.

<<interface>> SipStateMachine	**SIP Semantic**
`void doRequest(SipServletRequest req)`	The event is a SIP Request message.
`void doInitialRequest(SipServletRequest req)`	The event is an Initial SIP Request message.
`void doSubsequentRequest(SipServletRequest req)`	The event is a Subsequent SIP Request message.
`void doResponse(SipServletResponse resp)`	The event is a SIP Response message.
`void doProvisionalResponse(SipServletResponse resp)`	The event is a SIP Provisional Response message.
`void doSuccessResponse(SipServletResponse resp)`	The event is a SIP Successful Response message.
`void doRedirectResponse(SipServletResponse resp)`	The event is a SIP Redirection Response message.
`void doErrorResponse(SipServletResponse resp)`	The event is a SIP Error Response message.
`void doFinalResponse(SipServletResponse resp)`	The event is a SIP Final Response message.
`void timeout(ServletTimer timer)`	The event is a SIP Timeout message.

Table 9, SIP Semantics for <<interface>> SipStateMachine

The interface described in Table 9 does not provide methods for the most specific types of SIP Messages as specified in the SIP [11] and related specifications [21]. Ideally, all possible SIP Messages should be defined by this interface. This would fully define the type of event without having to parse additional properties such as a guard. It was decided to leave some level of abstraction as a complete interface would become too large and complex, and would have to be updated each time new SIP events are defined.

The interface is shown in Java syntax and is similar to the interface provided by the JSR-116 SIP Container. This should not be a surprise as both interfaces are designed to represent SIP functionality.

Although not shown in the SipStateMachine interface, the use of a "do nothing" action is also introduced. This is from an interface perspective defined by not implementing any method. From a State Chart model perspective this is defined by not drawing a transition in the model. The behavior of a "do nothing" actions is exactly as the name suggests. No action is performed and the control of the SIP dialog is returned to the SIP Container which will perform a default action.

It was observed that the use of an interface also allows describing the type of SIP events in more detail as this is inferred from the well defined interface. In addition, the structure of the model is impacted.

10.5 State Chart Model Design

The SIP Semantics are reflected through the use of transitions. All transitions defined in the model are now mapped to the interface. Seen from the UML formalism this means that the *operation* of a *callevent* of a *transition* is mapped to a method of the SipStateMachine interface.

The generic transitions of type SIP Request or SIP Response are still supported, but the use of specific methods is advised. In the end, it is the choice of the *architect* how to design the service in the model.

The use of an overlapping interface is reflected in the use of overlapping transitions. The design guideline from 6.3 disallowing overlapping transitions is dropped. The choice between overlapping transitions is now decided through priorities. Transitions with the same SIP Response type and overlapping guards are still disallowed as these cannot be prioritized.

SIP Response Type	**SIP Response Status Code**						**Priority**
	1xx	**2xx**	**3xx**	**4xx**	**5xx**	**6xx**	
Provisional Response							1
Successful Response							1
Redirection Response							1
Error Response							2
Final Response							3
SIP Response							4

Table 10, Mapping SIP Response Types and Status Codes

The State Diagram model may define overlapping transitions mapped to the various SIP Response methods. As a rule more specialized SIP Response methods have priority over more generalized methods. This is visualized by the Priority column in Table 10.

In addition to the mapping to the model interface, a transition can still define a guard. The guard, for SIP Response events, is validated against the SIP Response status code. As a rule, a SIP Response event for a certain SIP Response Type with a guard specified has priority over an event with the same type but with no guard set.

Use of these new design guidelines not only defines detailed SIP information in the model, but also provides a richer tool set for the *architect*. The State Chart fragments shown in Figure 44 and Figure 45 are identical in behavior but differ in description.

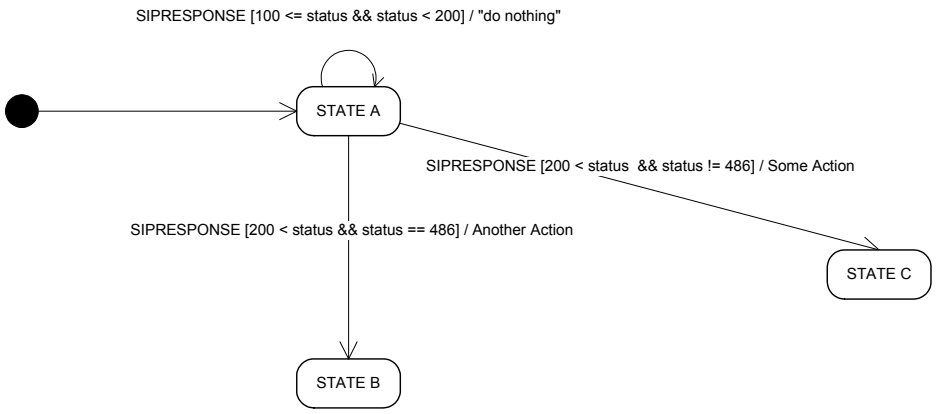


Figure 44, State Chart Fragment 1

Fragment 1 contains three transitions on SIP Response level in which the guards are carefully constructed to prevent overlap. Even though the transition going back to STATE A does nothing, it is still present in the model to prevent a gap of the covered SIP Response domain.

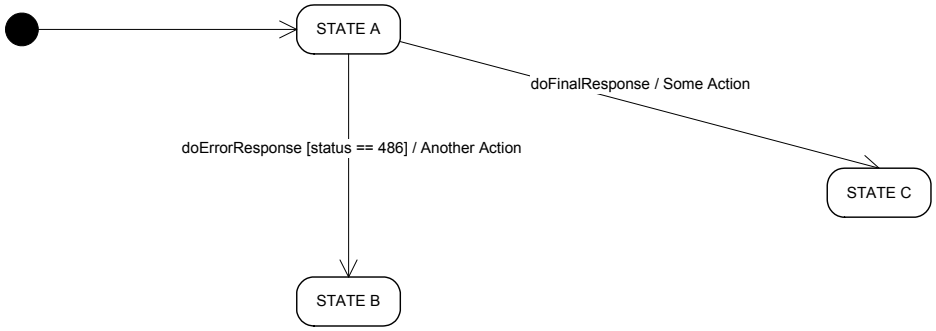


Figure 45, State Chart Fragment 2

Fragment 2 describes the same behavior with a reduced number of transitions. The transition from STATE A to STATE B has no guard set because the transition from STATE A to STATE C is more specific and has a higher priority. The "do nothing" transition is implied by not specifying a transition for the Provisional SIP Response domain. It could be a design choice to disallow implied transitions. In this case a transition mapped to the generic doResponse method and no guard could be added as a loopback to STATE A.

10.6 Java State Machine Structure and Generation

The use of an interface with prioritized methods impacts the design of the State Machine implementation. Two directions were identified to correct this.

- Updating the State Machine structure as described in chapter 8.8.2.

  This involves changing the State interface to conform to the SipStateMachine interface as used in the model, as well as adding logic to the SipStateMachine context class to map incoming events to the correct interface methods.

- Updating the concrete state implementations as described in chapter 8.8.2.

  This involves adding logic to all concrete state implementations to handle the various methods as defined by the SipStateMachine interface.

The logical choice from a technical point of view would be the first option. This centralizes the logic to handle incoming events in a single class. In addition, the state machine will still be based on a single interface, instead of some mix of two interfaces.

It was decided however to change the structure of the concrete states to handle the updated model. The purpose of this choice was to test whether the more complex concrete state could still be generated through Repleo.

This resulted in Repleo Templates that were heavy on iteration and conditional selection instructions. This resulted in some observations of the Repleo instruction set. The template files become complex, so some kind of Repleo comments would be useful. To insert a default code block the conditional selection instruction must check on the non existence of some data element. These instructions were initially not available and where requested to be added to the Repleo instruction set.

In the end it posed little difficulty to create a template file for the more complex concrete state. While the template file looks complex due to the large number of Repleo instructions, the design remains structured.

We were able to generate the Java boilerplate source code (see chapter 11) without any problems or errors. After manually implementing the action placeholders, the end to end test case was successfully executed

10.7 Conclusion and Remarks

The purpose of this analysis was to add additional SIP Semantics to the model. This information was expected to be useful for model design and template design for generation.

Two technologies were identified (Stereotypes and an Interface) and both can be used to add SIP Semantic information to the model. Stereotypes only add a description to the model while an interface impacts the structure. As the change to the structure of the model was seen as positive (see chapter 10.5) further analysis of SIP semantics was based on adding an interface.

A SIP Interface was introduced and used to map SIP *events*. Ideally, the SIP Interface used in the model should match the interface provided by the implementation structure. This reduces the complexity of the transformation as only a mapping between both interfaces should be generated.

As a design choice we used the SIP Interface in the model and added logic to the transformation to generate an implementation based on the regular SIP Container SIP Interface. We were able to perform this more complex transformation using Repleo. The resulting implementation passed the test cases.

It may be possible to introduce an external library of SIP *actions* accessible through an interface. This external library should contain implementations for the common SIP actions such as proxy, respond, or forward. This would allow automatic generation of implementations for actions, instead of placing descriptions in comments. This external library could be extended to include additional functionality.

In addition to an external library for SIP *actions*, libraries and corresponding interfaces could be used to include behavior outside of the standard SIP context.

11 Design Process

11.1 Introduction

The generation of source code based on some model is a single step in a larger process. This chapter describes the end to end design process for the SCE and Repleo based transformations. The scope of the process is limited to the actual actors, artifacts and transformation steps needed for the transformations.

This chapter uses the term *boilerplate implementation* to indicate an implementation (e.g. source code) which may be syntactical correct but is incomplete in the sense that it may cause errors or incorrect behavior when being deployed and executed. A further processing step is required to generate a *deployable implementation* which is a complete and deployable implementation.

The incomplete implementation is a result of limitations of the model. In particular, the information contained in the description of *actions* related to transitions is insufficient for automatic generation of source code implementing said action.

The total of the steps performed for the generation of source code could be referred to as an *automatic street* or *transformation street*.

11.2 SCE Design Process

The design process to create or generate a deployable implementation of a service specified as a list of requirements using SCE transformation involves three steps.

1 Create a State Chart model that complies with the service requirements.

2 Generate a boilerplate implementation from the model.

3 Generate a deployable implementation by completing the boilerplate implementation.

These steps are discussed in the following chapters.

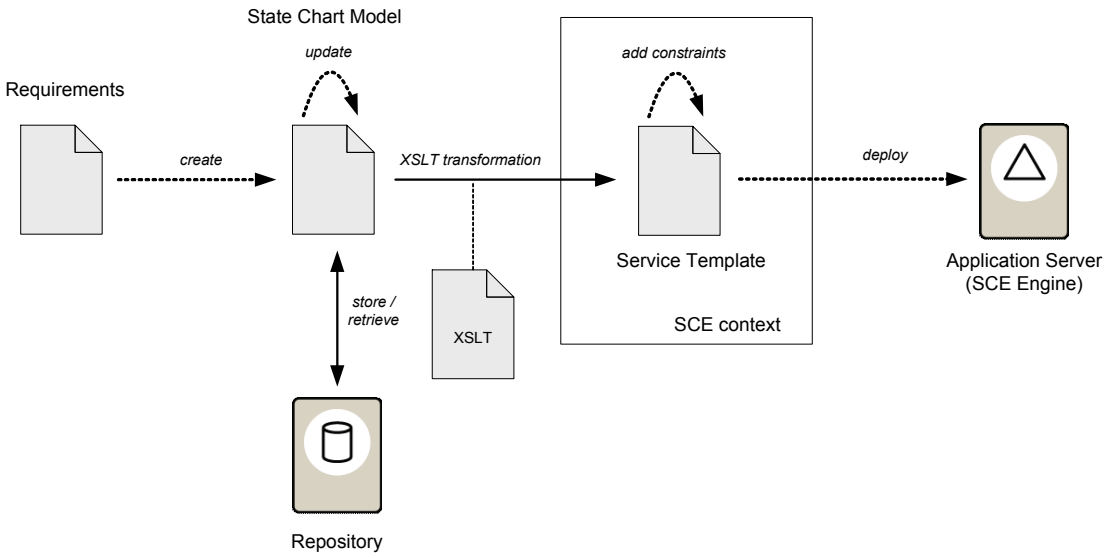


Figure 46, SCE Design Process

The model in Figure 46 describes the end to end design process for the SCE transformation. This figure shows the artifacts, nodes such as an Application Server or a Repository and tasks represented by arrows. An arrow with a dotted line indicates a task performed by a (human) actor. An arrow with a regular line indicates a task that is automated.

11.2.1 Create State Chart Model

The Requirements document represents the description of a service and is created outside of the described design process. The format is typical a textual description.

An *architect* creates the State Chart model based on the requirements document. The State Chart represents the service modeled as a state chart diagram. It is assumed the State Chart is created and updated using a specialized tool (e.g. a UML editor) that can export the state chart diagram in a well defined electronic format such as XMI.

The State Chart model is considered to be leading in design. It is the artifact that is considered to be up to date and which is stored in a repository. The generation is based on this artifact.

The state chart does not contain SCE specific bindings. That is, the binding of service constraints on Service Element and Composition Template level cannot be added during creation of the State Chart in the context of a UML editor as the required information for these bindings is only available in the SCE context. While the model may not contain the SCE specific binding for the actions, the semantics are still included through the use of a textual description (e.g. some kind of pseudo code).

A compliant XMI document does not contain tool specific information such as the location of diagrams. The document in which the State Chart is stored in the repository is typically a tool specific format. This introduces limitations when transforming back from a SCE Service Template into a XMI document.

11.2.2 Generate Boilerplate Implementation

The SCE Transformation transform a State Chart model represented as a XMI document into boilerplate source code. The source code is in a XML based format supported by the SCE IDE.

The transformation is an automated process. The transformation is based on an XSLT document. These XSLT documents are static and highly specific for this particular transformation.

Developing these transformation specifications introduces a new actor (e.g. a *tool engineer*) that combines aspects from an architect as well as a designer.

While the generated SCE Template describes the same service as specified in the State Chart model it is assumed a transformation back from the XML document into a XMI document will not be possible. For one, the SCE Template structure may be difficult to parse. Also, the original XMI document contains quite a bit of overhead that is lost during the transformation.

While the generated XML document that represents a syntactical valid Service Template which could be deployed on the SCE Engine, it misses the SCE specific binding.

11.2.3 Generate Deployable Implementation

The generation of a deployable SCE Template is a manual task performed by a *designer* using the SCE IDE. The task involves replacing the service descriptions that are generated as a descriptive text with actual service constraints. These SCE specific binding depends on the configuration of the SCE IDE and Engine. The SCE IDE provides features to view and select available constraints.

From a technical point of view, the constraints as such could be added to the State Chart model, and with this information available, the transformation could immediately generate a deployable SCE Template. However, the UML editor has no knowledge of the exact syntax of constraints, nor of the available constraints. Adding SCE specific binding immediately from the UML editor would require some kind of mechanism to lookup available constraints from the SCE context.

The boilerplate SCE Template represents a state machine implementation. While the structure is strongly hierarchical and not very complex, the size of a generated SCE Template, especially for larger State Charts, can become very large. The SCE IDE is based on a visual representation of the SCE Template and larger models are not completely visible. This not only introduces a usability penalty, but also increases the risk for errors due to the reduced overview.

The deployment, performed by the *deployer*, of Service Templates is not further discussed in this thesis.

11.2.4 SCE Design Process Limitations

The SCE Transformation design process as discussed in the previous chapters has a number of limitations.

- The State Chart model does not contain SCE specific bindings. The service constraints are only known from within the SCE context.

- Adding these service constraints using the SCE IDE is not intuitive. The generated boilerplate SCE Template is too large and complex.
- Information regarding the added service constraints is not stored in the repository.

Two directions were identified to reduce these limitations.

First, the UML editor could be customized to allow access to the SCE context. For example, a plug-in could be created that can access the SCE database and provides view and select functionality similar to the SCE IDE. This solves all three limitations but requires adaptation of the UML editor.

Secondly, the SCE IDE could be enhanced by adding some UML editing functionality. The UML State Chart could be loaded into the SCE IDE and represented as a State Chart Diagram or State Chart Table. The service constraints are then added to the model as the SCE specific binding is available from within the SCE context. A Service Template is then generated from the updated model. This solves problem 1 and 2.

11.3 Repleo Design Process

The design process to create or generate a deployable implementation of a service as specified as a list of requirements using Repleo generation is similar to the SCE transformation design process in that it involves three similar steps.

1 Create a State Chart model that complies with the service requirements.

2 Generate a boilerplate implementation from the model.

3 Generate a deployable implementation by completing the boilerplate implementation.

Besides the different scope (the Repleo transformation generates Java source code instead of a SCE Template) there are also differences in the reason to (also) first generate a boilerplate implementation or in the tasks assigned to the actors.

The steps are discussed in the following chapters.

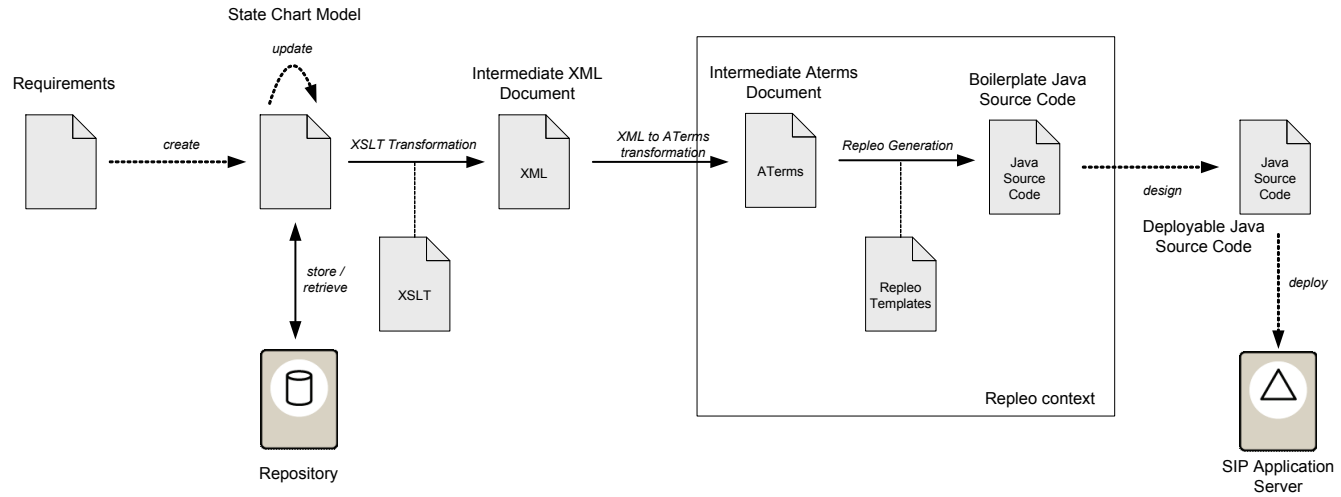


Figure 47, Repleo Design Process

The model in Figure 47 describes the end to end design process for the Repleo transformation. The rectangle indicating the Repleo Context is shown for the Repleo Generation transformation step. The pre-processing steps (XSLT Transformation and XML 2 ATerms transformation) could be considered to be part of the Repleo Context as well.

11.3.1 Create State Chart Model

As with the SCE Transformation the State Chart model is created by the *architect* based on the requirements and is considered to be the leading design document. The generation is based on this model.

The inability to include SCE specific bindings introduced an extra transformation step from a boilerplate implementation to a deployable implementation. As the target implementation is Java source code the service constraints are not in the scope. However, there is still a problem describing the service behavior in the model.

For the SCE transformation, the service constraints operated as an interface to a (not further described) library. The task of the *designer* was only to provide a mapping from a descriptive text to a constraint (e.g. a library reference) already available in the SCE context.

Such a library is not (readily) available for a Java implementation. As such the State Chart model does not contain (textual descriptions of) service constraints, but rather a textual description of the required action itself (or a reference for this action to some companion design document) to be used by the *designer* later in the process.

11.3.2 Generate Boilerplate Implementation

Due to the expected input data format of Repleo, some pre-processing steps are required. From a design process view these pre-processing steps and the Repleo generation step can be seen as a single transformation step.

The Repleo Transformation transforms a State Chart model represented as a XMI document into boilerplate source code. The source code is valid Java language source code.

As with the SCE design process, the transformation is an automated process. The XSLT and Repleo Template documents that specify the transformation are static and highly specific for this particular transformation. Again, a dedicated actor is assumed for development of these documents.

The generated Java source code is syntactical correct and could even successfully be deployed on an Application Server. But as it lacks the implementation of the actions the behavior would not be in line with the requirements.

11.3.3 Generate Deployable Implementation

The generated boilerplate Java implementation is incomplete. The implementations of the actions are missing. Based on the description in the State Chart model, the generation does include comments, at the correct places in the generated source code, describing the action to be performed.

For the SCE transformation, the *developer* only had to include a mapping using the SCE IDE, for the Repleo transformation, the actual implementation has to be developed. This would typical require a full Java based IDE such as the SDS platform [18].

In contrast with the SCE transformation in which (the structure of) the generated boilerplate implementation is hardly changed, it is expected the generated Java source code is primarily used by the *designer* as a template to base the deployment implementation on.

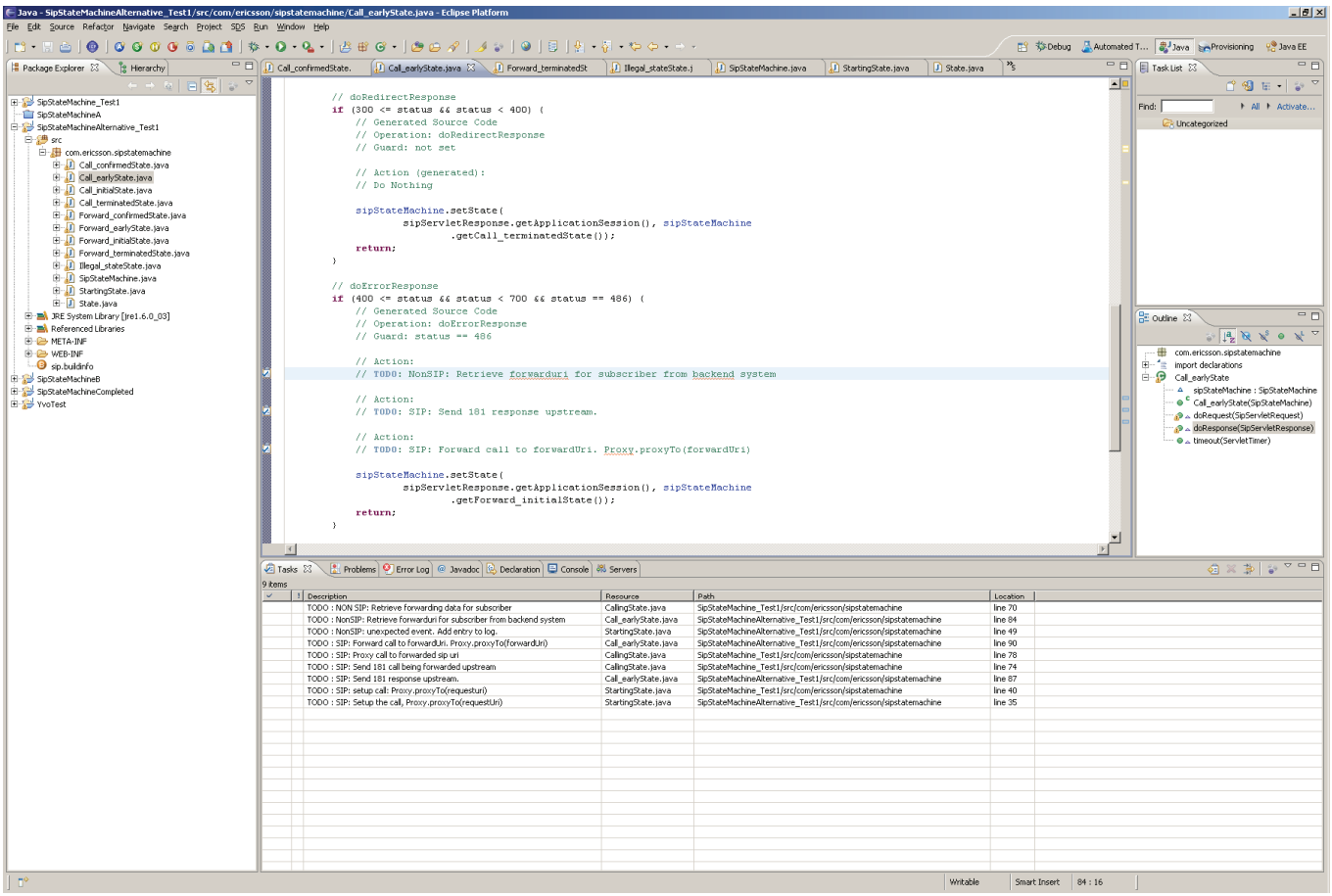


Figure 48, SDS 4.0 Environment with Generated Source Code

The image in Figure 48 shows a screenshot of the SDS 4.0 design environment with a fragment of generated source code visible. The comments with a description of the action are clearly visible. A designer needs to replace these comments with an actual implementation

The deployment, performed by the *deployer* is not further discussed in this thesis.

11.3.4 Repleo Design Process Limitations

The Repleo Transformation design process as discussed in the previous chapters has a number of limitations.

- An additional development step is needed due to limitations in the model.
- Development on the boilerplate implementation is not stored in the repository.

Different approaches are possible to tackle these problems.

- A secondary repository could be used to store the deployable implementation together with a framework to combine newer boilerplate implementations with stored deployable implementations.

- Similar to the SCE transformation the implementation of the actions could be abstracted as a library with an interface that can be included in the State Chart model. (See also chapter 10.)

- Enhance the State Chart model through the use of pseudo code from which Java source code can be generated, or directly use Java source code in the model. This would require a specialized combined Model and Source Code design environment.

It is fairly easy to setup a transformation from a State Chart model into a boilerplate Java implementation that can be used as a template by a designer. However, using the model as the main design artifact poses both technical as process difficulties.

12 Conclusion and Remarks

12.1 Introduction

This chapter provides conclusions on the investigation in general and should be seen in addition to the SCE Transformation, Repleo Transformation and SIP Semantics specific conclusions in chapter 7.5, chapter 9.6 and chapter 10.7.

12.2 Conclusions

The question whether support for state machines can be provided in SCE, as stated in the problem description in chapter 2.5 is partially answered.

We demonstrated a Composition Template design that utilizes a state machine structure. Although the behavior of the Composition Template could not be verified, we believe a state machine structure can be designed using the SCE DSL.

However, the Composition Template cannot be executed due to not implemented features in the SCE Engine. We note that the implementation of these missing features may be technically challenging. In addition, the lack of event handling in SCE would prevent deployment of such a Composition Template in a production environment.

As an alternative, support for state machines was investigated for JSR-116 based SIP Applications.

In this investigation we demonstrated, with the Repleo State Machine Transformation, an end to end transformation from a UML State Chart Diagram into deployable Java source code that had the same behavior as the reference Java source code implementation.

However, a number of limitations were identified in the transformation and are listed below.

- The behavior of the IMS Service is not fully represented in the IMS Model. The *actions* that are part of the IMS Service are difficult to describe, with regard to automatic generation. A manual design step was required to generate Deployable Java Source Code.

- Theoretical, the model should be implementation independent. However, this adds to the complexity of the transformation as additional transformation steps from a neutral pseudo language into the implementation language are needed.

- The transformation process itself is inflexible. Any change to the structure of the generated source code requires rework on the transformation process.

- The model does not reflect any adaptation made to the generated source code after the transformation. In effect, these adaptations are lost during following transformations.

- The design of the Java State Machine implementation is based on the standard State Pattern. This design is insufficient for a production level

implementation. The State Pattern structure is not optimal for an Application Server environment, and features such as synchronization or exception handling are missing. We estimate that the development of such a reference production level implementation itself is a major undertaking.

To address these issues, a number of future research topics are recommended in chapter 12.4.

Due to the introduction of the Engineering Assignment, the questions on the SCE service model and Component Frameworks, as stated in the Problem Description, are only partially answered. The SCE Service Model is described in chapter 4. Component Frameworks as such are not investigated. The use of an external library referenced through an interface from within the model, as discussed in chapter 10.7, could be considered to be an alternative component framework.

12.3 Remarks

A transformation from a model into an implementation is in itself insufficient to reach the goal of more flexibility and faster Time To Market as stated in the problem description in chapter 2.5.

- In addition to a (technical) transformation extensive work is needed on the specification of an IMS metamodel, choice and customization of tooling and process.

- While transformation adds flexibility as it allows changes in the IMS Model to automatically propagate into the implementation, it also reduces flexibility as the structure of the implementation becomes static. The transformation street itself is inflexible.

Due to the overhead to setup a transformation street, it only makes sense for generation of a larger number of similar IMS Services.

Adaptation of a full Model Driven Engineering process for software development has a large impact on an existing design process. Tooling, processes, repositories, documentation, roles are all affected.

We see a practical use for transformations though. Boilerplate source code generation itself, without the goal of incorporating frequently changing models may already be valuable for development.

12.4 Future Work

12.4.1 Investigate an IMS Metamodel

The behavior of the IMS Service is not fully represented in the IMS Model. In particular, the *actions* that are part of the IMS Service are difficult to describe.

Chapter 10 describes the use of an Interface to represent SIP events. Such an Interface could also be used to reference actions that are provided in an external library.

- Can a library and interface be created that implement a set of standard SIP actions such as proxy, send reply, invite, or to operate as a B2BUA?

- What kind of non SIP actions can be identified? Relevant areas could be accessing subscriber data, interfacing with customer backend systems, or utilizing web services.

- What kind of framework should be used for these libraries? Consider design, process, tooling, deployment, and maintenance issues.

12.4.2 Investigate a SIP State Machine Framework

The Java State Machine implementation used in this thesis is based on the standard State Pattern. This design is insufficient for a production level implementation.

- Investigate a SIP State Machine Framework. What are the requirements for such a framework? What are the non functional requirements for such a framework such as performance, stability, or testability?

- Should such a framework be designed in house or are third party products available?

- Is the structure usable as a generation target?

12.4.3 Investigate the Setup of a Transformation Street

The transformation street for the Repleo State Machine Transformation, as described in chapter 11, includes several transformation steps and technologies.

- Investigate the setup of a transformation street for a professional design environment in which IMS Services are developed.

- Identify the requirements for such an environment. Consider both technical and process related requirements.

- What kinds of tools are needed? Are these available from third party vendors? Is a customization needed?

- Implement a prototype of said transformation street.

- Does the existing design process need to be changed? Are roles of stakeholders changed or are new roles required?

- How flexible is the transformation street? What tools or processes are available to assist with design of the transformation street?

12.4.4 Validation

In this thesis an IMS Model was transformed into an executable implementation. The model could also be used for validation of the IMS Service.

- Investigate what kind of validation can be performed based on a model of an IMS Service?

- Does the IMS Model need to be extended to perform validation?

Preliminary investigation suggests validation could be performed on the model itself.

- Properties of fields such as the SIP Request Method type could be validated.

- The behavior the IMS (State Chart) Model could be compared to the State Machine behavior implied by the SIP specification.

In addition, System Management sees possibilities for validation of the runtime behavior of the IMS Service. The IMS Service is considered to be a black box. Incoming and outgoing events are intercepted and validated against the behavior described in the IMS Model.

13 Glossary

3GPP	3rd Generation Partnership Project
AN	Access Network
AS	Application Server
ASF	Algebraic Specification Formalism
ATF	Automatic Testing Framework
B2BUA	Back-to-Back User Agent
BMUM	Business Unit Multimedia
CAMEL	Customized Applications for Mobile network Enhanced Logic
CDIV	Communication Diversion
CFB	Call Forwarding on Busy
CN	Core Network
CS	Circuit Switched
DMMP	Development Unit Multimedia Products
DSL	Domain Specific Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Design Environment
IETF	Internet Engineering Task Force
IFC	Initial Filter Criteria
IMS	IP Multimedia Subsystem
ISDN	Integrated Service Data Network
J2EE	Sun Java Enterprise Edition
Java EE	Java Platform, Enterprise Edition
JCP	Java Community Process
JSR	Java Specification Request
MSA	Multi Service Architecture
MSISDN	Mobile Station Integrated Services Digital Network
MTEL	Multimedia Telephony
MU	Market Unit

NSM	Network System Modeling
OMG	Object Management Group
OSA	Open Service Access
PDU	Product Development Unit
PLCM	Product Life Cycle Management
PLMN	Public Land Mobile network
PSTN	Public Switched Telephony Network
PUI	Public User Identity
PSI	Public Service Identity
QOS	Quality of Service
PSTN	Public Switched Telephony Network
RTP	Real-Time Protocol
RTSP	Real-Time Streaming Protocol
SAR	Servlet Archives
SCE	Service Composition Environment
SCS	Service Capability Server
SDF	Syntax Definition Formalism
SDP	Service Description Protocol
SIP	Session Initiation Protocol
TTM	Time To Market
UAC	User Agent Client
UAS	User Agent Server
UE	User Equipment
UML	Unified Modeling Language
URI	Uniform Resource Identifier
VAS	Value Added Services
XMI	XML Metadata Interchange
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

14 References

[1] Student Assignment, ETM/R/E-05:0059 Uen_PA7,

[2] Detailed Student Assignment, ETMWOBR, Uen_PA7

[3] TISPAN; Multimedia Telephony with PSTN/ISDN simulation services, ETSI TS 181 002 V1.1.1

[4] TISPAN; PSTN/ISDN simulation devices: Communication Diversion (CDIV); Protocol specification, ETSI TS 183 004 v 1.2.1

[5] XSL Transformations (XSLT), Version 1.0, W3C Recommendation 16 November 1999, World Wide Web Consortium, http://www.w3.org/TR/xslts

[6] XML Path Language (XPath), Version 1.0, W3C Recommendation 16 November 1999, World Wide Web Consortium, http://www.w3.org/TR/xpath

[7] OMG Unified Modeling Language, Specification, Version 1.4, September 2001, Object Management Group, http://www.omg.org/spec/UML/1.4/

[8] OMG XML Metadata Interchange (XMI), Version 1.2, Object Management Group, http://www.omg.org/spec/XMI/

[9] Extensible Markup Language (XML) 1.0 (Fourth Edition), W3C Recommendation 16 August 2006, World Wide Web Consortium, http://www.w3.org/TR/xml/

[10] Technical Specification Group Services and System Aspects; IP Multimedia Subsystem (IMS); Stage 2 3GPP TS 23.228 R6

[11] SIP: Session Initiation Protocol RFC3261, Internet Engineering Task Force, Network Working Group, June 2002

[12] The Java Community Process, JSR-116: SIP Servlet API, http://jcp.org/en/jsr/detail?id=116

[13] The Java Community Process, JSR-289: SIP Servlet v1.1, http://jcp.org/en/jsr/detail?id=289

[14] Project GlassFish, https://glassfish.dev.java.net/

[15] Project SailFin, https://sailfin.dev.java.net/

[16] Design Patterns: elements of reusable object-oriented software / Erich Gamma .. [et al.], Addison-Wesley, ISBN 0-201-63361-2

[17] ECharts for SIP Servlets,
http://echarts.org

[18] IMS application development tool - SDS 4.0
Ericsson Mobility World, Developer Program
http://www.ericsson.com/mobilityworld

[19] Eclipse, an open development platform,
Eclipse Foundation
http://www.eclipse.org/

[20] The ASF+SDF Meta Environment,
Centrum voor Wiskunde en Informatica (CWI)
http://www.asfsdf.org

[21] Change Process for the Session Initiation Protocol (SIP),
RFC 3427, Internet Engineering Task Force, Network Working Group,
December 2002

[22] Jeroen Arnoldus and Jeanot Bijpost and Mark van den Brand,
Repleo: a syntax-safe template engine,
In GPCE '07: Proceedings of the 6th international conference on
Generative programming and component engineering, pages 25--32,
2007. ACM.

[23] Brand, M.G.J. van den and P. Klint (2007),
"ATerms for manipulation and exchange of structured data: It's all
about sharing.",
Information and Software Technology 49:55--64.

[24] Stratego, Specification of Program Transformation Systems,
http://www.cse.ogi.edu/pacsoft/projects/Stratego/

[25] Java API for XML Parsing (JAXP),
https://jaxp.dev.java.net/

[26] http://www.program-transformation.org/Tools/ATermToXML

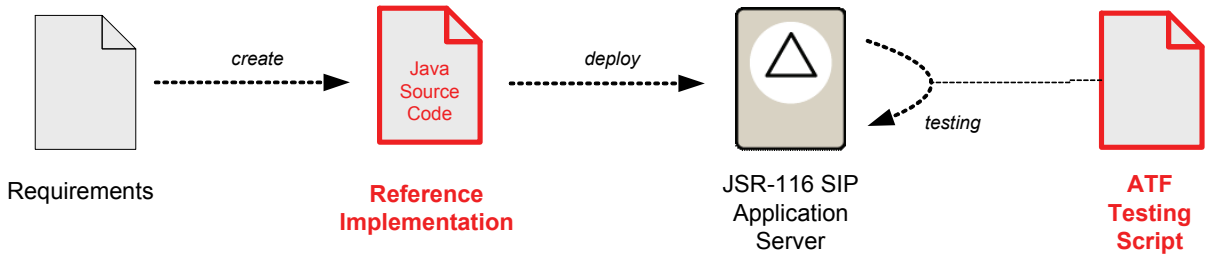
Appendix A Overview of Documents and Source Code

This appendix contains an overview of the documents and source code created during the investigation. These documents are available on the *Thesis Companion CDROM*, or by contacting the author through e-mail.

- `mailto: thesis@meneerbruggeman.nl`

Java Reference Implementation

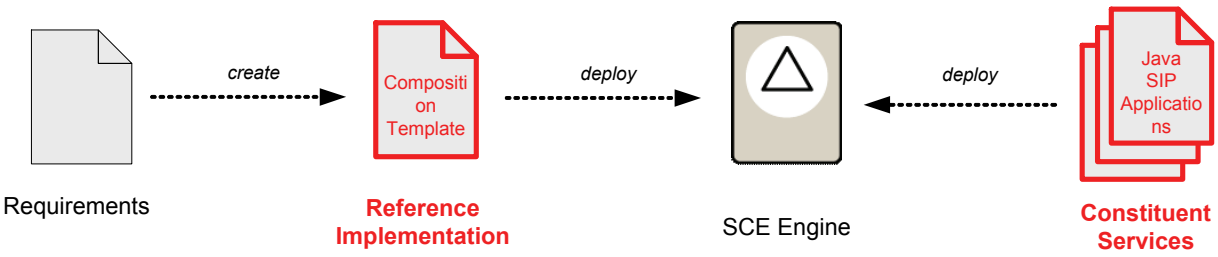
This is the Java JSR-116 reference implementation of the Call Forwarding on Busy service.



- `/JSR116_Reference/`
  - The Java source code for the regular design
  - The compiled and packaged sar container
- `/JSR116_Reference_StateMachine/`
  - The Java source code for the state machine design
  - The compiled and packaged sar container
- `/ATF/`
  - SDS 4.0 ATF Testing Script for Call Forwarding on Busy

SCE Reference Implementation

This is the SCE Composition Template reference implementation of the Call Forwarding on Busy service.



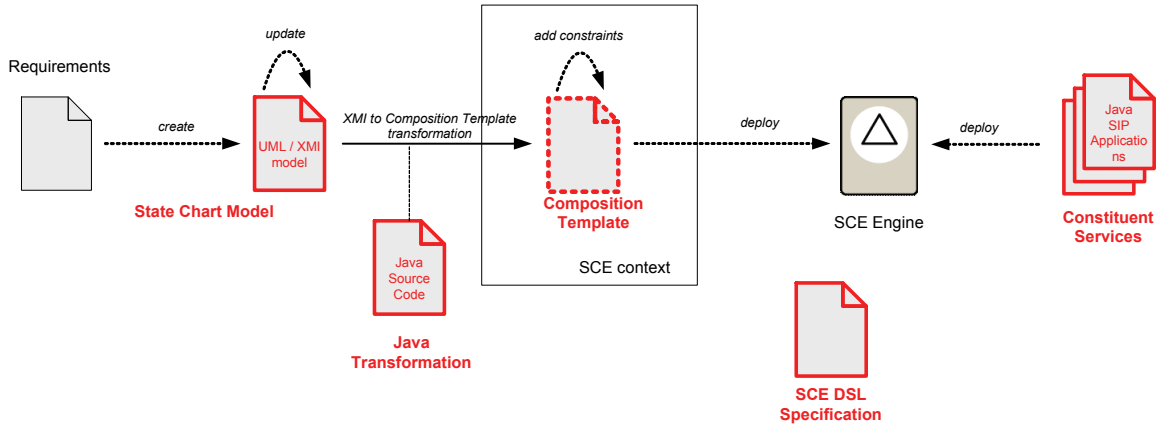
- `/SCE_Reference/`
  - The Composition Template for the regular design (as XML document)
- `/SCE_Reference_StateMachine/`

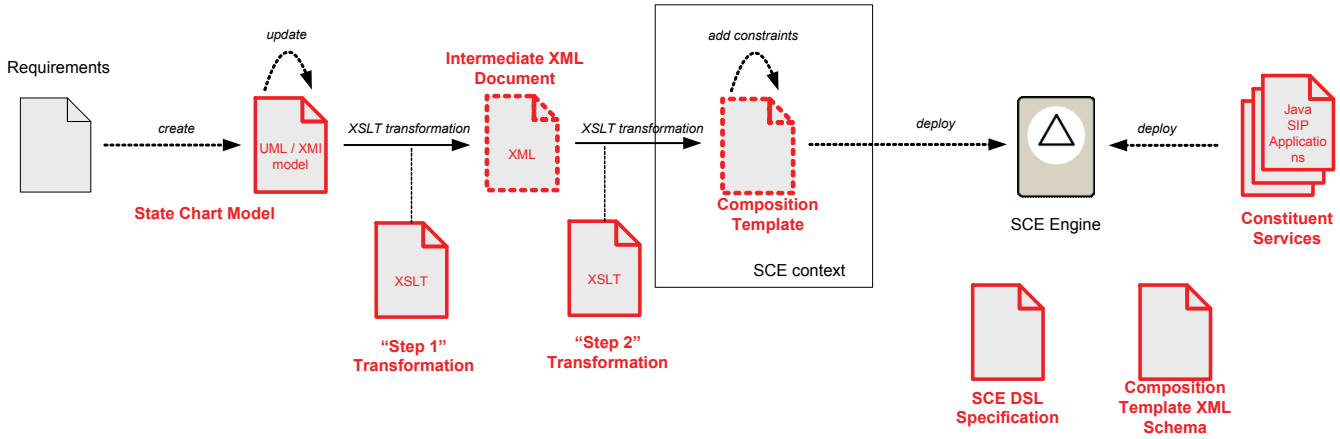
- The Composition Template for the state machine design (as XML document)

- `/SCE_Constituent_Service/`
  - The Constituent Services used by the Composition Templates

SCE Transformation

This is the SCE Transformation from a UML State Chart Model into a Composition Template, including supporting Constituent Services. A Java based transformation and a XSLT based transformation is available.





- `/SCE_Transformation_JAVA/`
  - The Java SCE Transformation source code

- `/SCE_Transformation_Model/`
  - The CFB State Chart Model as UML Model
  - The CFB State Chart Model as XMI Document

- `/SCE_Transformation_XSLT`
  - The "step 1" XSLT specification from XMI Document to Intermediate XML Document
  - The "step 2" XSLT specification from Intermediate XML Document to XML Composition Template Document

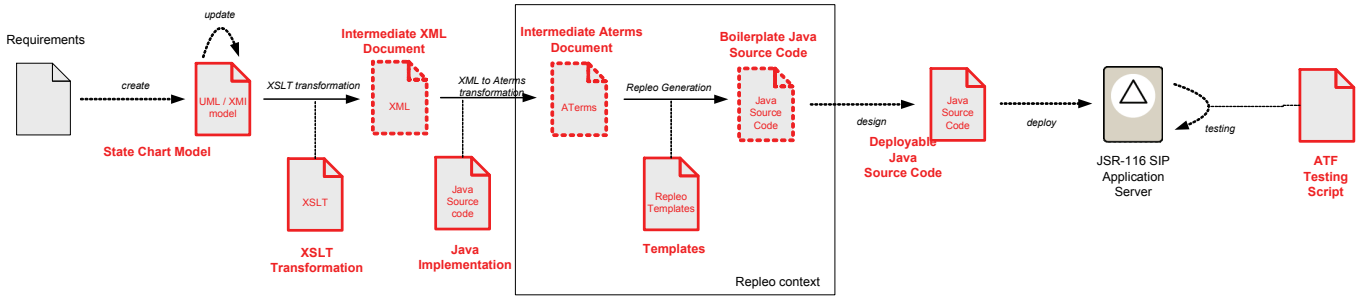
- Example of the generated Intermediate XML Document
- Example of the generated Composition Template

- `/SCE_Constituent_Service/`
  - The Constituent Services used by the Composition Templates

- `/SCE`
  - The Composition Template XML Schema

Repleo Transformation

This is the Repleo Transformation from a UML State Chart Model into Deployable Java Source Code. The generated SIP Application can be packaged and tested using the SDS Automatic Testing Framework.



- `/Repleo_Transformation_Model`
  - The CFB State Chart Model as UML Model
  - The CFB State Chart Model as XMI Document

- `/Repleo_Transformation_XSLT`
  - The XSLT specification from XMI Document to Intermediate XML Document
  - Example of the generated Intermediate XML Document

- `/Repleo_Transformation_Java`
  - The Java source code for the transformation from a XML Document into an ATerms Document
  - Example of the generated ATerms Document

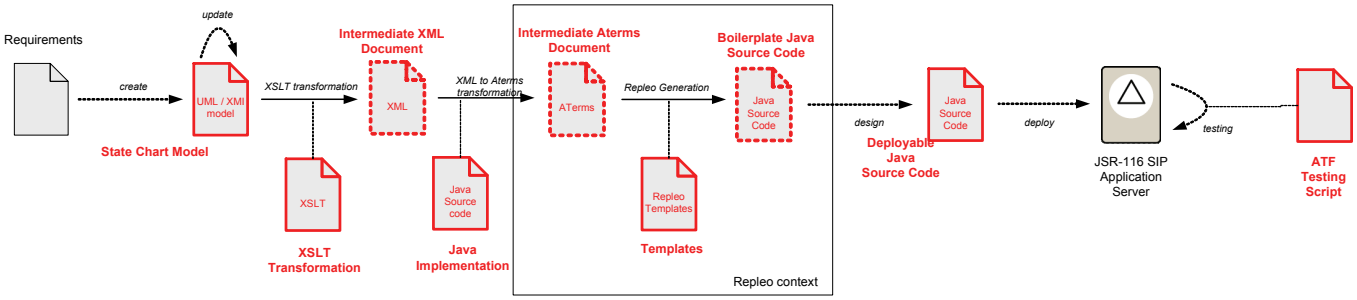
- `/Repleo_Transformation_Templates`
  - The Repleo Templates for the generation of Boilerplate Java Source Code
  - Example of the generated Boilerplate Java Source Code
  - Example of the Deployable Java Source Code
  - The compiled and packaged sar container

- `/ATF/`

- SDS 4.0 ATF Testing Script for Call Forwarding on Busy

SIP Semantics and Update Repleo Transformation

This is the updated Repleo Transformation from the enhanced UML State Chart Model into Deployable Java Source Code. The generated SIP Application can be packaged and tested using the SDS Automatic Testing Framework.



- `/Repleo_Transformation2_Model`
  - The enhanced CFB State Chart Model as UML Model
  - The enhanced CFB State Chart Model as XMI Document
- `/Repleo_Transformation2_XSLT`
  - The XSLT specification from XMI Document to Intermediate XML Document
  - Example of the generated Intermediate XML Document
- `/Repleo_Transformation2_Java`
  - The Java source code for the transformation from a XML Document into an ATerms Document
  - Example of the generated ATerms Document
- `/Repleo_Transformation2_Templates`
  - The Repleo Templates for the generation of Boilerplate Java Source Code
  - Example of the generated Boilerplate Java Source Code
  - Example of the Deployable Java Source Code
  - The compiled and packaged sar container
- `/ATF/`
  - SDS 4.0 ATF Testing Script for Call Forwarding on Busy