A fine-grained approach towards asynchronous service composition of heterogeneous services

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der RWTH Aachen University zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Konstantinos Vandikas
M.Sc.

aus Kozani, Griechenland

Berichter: Universitätsprofessor Dr. rer. pol. Matthias Jarke

Universitätsprofessor dr. ir. Mike Papazoglou

Universitätsprofessor Dr. Wolfgang Prinz

Tag der mündlichen Prüfung: 3. December 2014

Abstract

In software design, a service-oriented architecture is a set of principles and methodologies used for designing and developing software in the form of interoperable services. Each service encapsulates well-defined business functionality and it is built as a reusable component. Thereafter, new services can be generated as a coordinated aggregate of pre-existing functionality by means of service composition.

Common practice in the Information and Communication Technology domain (ICT) is the usage of standardized workflow languages in order to describe the interaction between such services. Examples of such languages are the Web Services Business Process and Execution Language (WS-BPEL) and the Business Process Modeling Language (BPMN). At runtime, a framework interprets the workflow and performs the actions mandated by the semantics of its constructs. Even though, a workflow language contains a sufficient amount of constructs to qualify as Turing complete, the usage of existing workflow languages along with their corresponding frameworks renders them cumbersome for rapid application development where one needs to combine services from heterogeneous domains and in particular when re-using pre-existing services originating from the telecommunications domain.

More specifically, the limitations in the state of the art for workflow languages are encountered in aspects such as tight-technological coupling; interaction is limited to particular technologies, usage of static type systems - that hinder experimentation and finally yet importantly in terms of parallelism and concurrency, where the designer of a workflow is forced to manually define execution order in an attempt to utilize multiple cores which are commonly found in most computer systems nowadays.

This dissertation introduces a novel language for service composition and a technology agnostic composition framework suitable for developing and executing service compositions of heterogeneous services. The proposed service composition language is concurrent by default; parallel execution of actions is determined by their corresponding data dependencies. The proposed framework allows for an optional type system permitting both typed and un-typed variables. Un-typed variables can be used while designing and experimenting with the composition in a trial and error fashion; while typed can be used once the model of the service composition matures and becomes production-ready. Moreover, the proposed composition framework employs a fine level of granularity while interpreting the constructs of the proposed language.

Our qualitative evaluation of the proposed language has shown that it is capable of expressing a wide set of workflow patterns, making it as expressive as rival workflow languages. Empirical evaluations of the proposed fine-grained composition framework have shown that is scalable; limited only by the amount of available memory and not by the number of available processing threads.

Kurzfassung

Beim Software-Design bezeichnet man als Service-orientierte Architektur (SOA) die Prinzipien und Methoden, die beim Design und Entwicklung vom Software in Form von interagierenden und interoperablen Diensten benutzt werden. Jeder Dienst kapselt eine bestimmte Funktionalität und wird als eine wiederverwendbare Komponente gebaut. Basierend darauf können neue Dienste als koordinierte Aggregationen aus schon vorhandenen Diensten mit Hilfe von Service Composition generiert werden.

Die gängige Praxis ist die Nutzung von standardisierten Workflow-Sprachen, um die Interaktion zwischen solchen Diensten zu beschreiben. Einige Beispiele von solchen Workflow-Sprachen sind Web Services Business Process and Execution Language (WS-BPEL) und Business Process Modeling Language (BPMN). Das Framework interpretiert Workflows zur Laufzeit und führt die Aktionen aus, so wie sie durch die Semantik von verschiedenen Sprachkonstrukten vorgegeben sind. Obwohl eine Workflow-Sprache oft genug Konstrukte hat, dass sie als Turing-vollständig gelten könnte, zeigt die Nutzung von existierenden Workflow-Sprachen samt mit ihren entsprechenden Frameworks, dass sie relativ schwerfällig für schnelle Entwicklung von solchen Applikationen sind, wo man Dienste aus verschiedenen heterogenen Domänen kombiniert, besonders wenn existierende Dienste aus der Telekommunikationsdomäne wiederverwendet werden sollten.

Unter genauer Betrachtung vom Stand der Technik bestehen die meisten Einschränkungen von Workflow-Sprachen bei Aspekten wie: enge technologische Kopplung, Interaktion zwischen Diensten limitiert nur auf bestimmte Technologien, und Nutzung von statischen Typ-Systemen, wodurch das Experimentieren mit neuen Diensten und Applikationen erschwert wird.

Ebenfalls sehr wichtig sind die Einschränkungen im Bereich von Parallelität und gleichzeitiger Ausführung von Workflows, wo der Workflow-Designer gezwungen wird, die Reihenfolge der Ausführung eines Workflows manuell zu definieren, um mehrere CPU-Cores von heutigen Rechnersystemen effektiv auszunutzen.

Diese Dissertation stellt eine neue Sprache zur Servicekomposition und ein dazu passendes technologie-unabhängiges Framework für die Entwicklung und Ausführung von Servicekompositionen aus heterogenen Diensten vor. Die vorgeschlagene Sprache ist standardmässig parallel; die parallele Ausführung von Aktionen wird durch ihre Datenabhängigkeiten bestimmt. Das Framework beinhaltet ein optionales Typsystem, so dass sowohl typisierte als auch nicht-typisierte Variablen benutzt werden können. Die Variablen ohne Typ können z.B. beim anfänglichen Design und Experimentieren mit neunen Kompositionen benutzt werden. Die typisierten Variablen können dagegen später eingesetzt werden, wenn das Model von der Service Komposition sich stabilisiert und bereit für den produktiven Einsatz wird. Darüber hinaus unterstützt das vorgeschlagene Kompositionsframework eine sehr feine Granularität beim Interpretieren von Konstrukten der vorgeschlagenen Sprache.

Die qualitative Evaluierung der vorgeschlagenen Sprache hat gezeigt, dass sie in der Lage ist, eine breite Palette von Workflow Patterns auszudrücken, mit den oben genannten Vorteilen gegenüber existierenden Workflowsprachen. Eine empirische Analyse und Evaluierung von dem vorgeschlagenen Composition Framework zeigt, dass das Framework skalierbar ist; eingeschränkt nur durch die Hauptspeicher-Grösse und nicht durch die Anzahl von den zur Verfügung stehenden Bearbeitungsthreads.

(Translation from the English abstract done by Roman Levenshteyn)

Στη μνήμη των γονιών μου
(In loving memory of my parents)

Acknowledgments

A great number of people helped me bring this research to fruition. First of all I would like to thank Prof. Dr. Matthias Jarke from the Chair of Information Systems (Informatik 5), RWTH Aachen University, Germany for his support and the freedom in research. Moreover, I would like to thank Prof. Mike Papazoglou from Tilburg University, the Netherlands and Prof. Wolfgang Prinz from the Chair of Information Systems (Informatik i5), RWTH Aachen University, Germany for being the second readers to this thesis.

I am grateful to my colleagues in Ericsson GmbH - Eurolab R&D, Germany. Especially I would like to thank Jörg Niemöller, Roman Levenshteyn, Raphael Quinet, Ioannis Fikouras and Friedhelm Ramme for the very long, sometimes spirited, but otherwise fruitful discussions we had.

Moreover, my family for supporting me in more ways than I could have thought were possible.

Last but not least, I would like to thank Maria for being there and encouraging me to see through the completion of this work.

Konstantinos Vandikas Aachen, November 3, 2013

Contents

Chapter 1

Introduction

In this chapter we describe the motivation for undertaking this research by describing the landscape for this work and its limitations. Thereafter we disclose the contributions made by this thesis along with a list of constituent papers where parts of the research detailed in this thesis have been published. We conclude this chapter by providing an outline of the thesis as a road map for our reader.

1.1 Landscape

In software design, a Service Oriented Architecture (SOA) is a set of principles and methodologies used for designing and developing software in the form of interoperable services. Each service encapsulates well-defined business functionality and it is built as a reusable component. Thereafter, new services can be generated as a coordinated aggregate of pre-existing functionality by means of service composition. Even though the process of service composition is as such protocol agnostic, it has been most successfully embraced by workflow languages and the corresponding execution frameworks that are limited to interact with Simple Object Access Protocol (SOAP) based services. Such frameworks aim at providing production grade quality and therefore employ static type systems.

However, as stated in Figure 1.1, today's service landscape consists of many different technologies used to implement services. This is a result of a historical development process aiming to find solutions for concrete technical problems within multiple

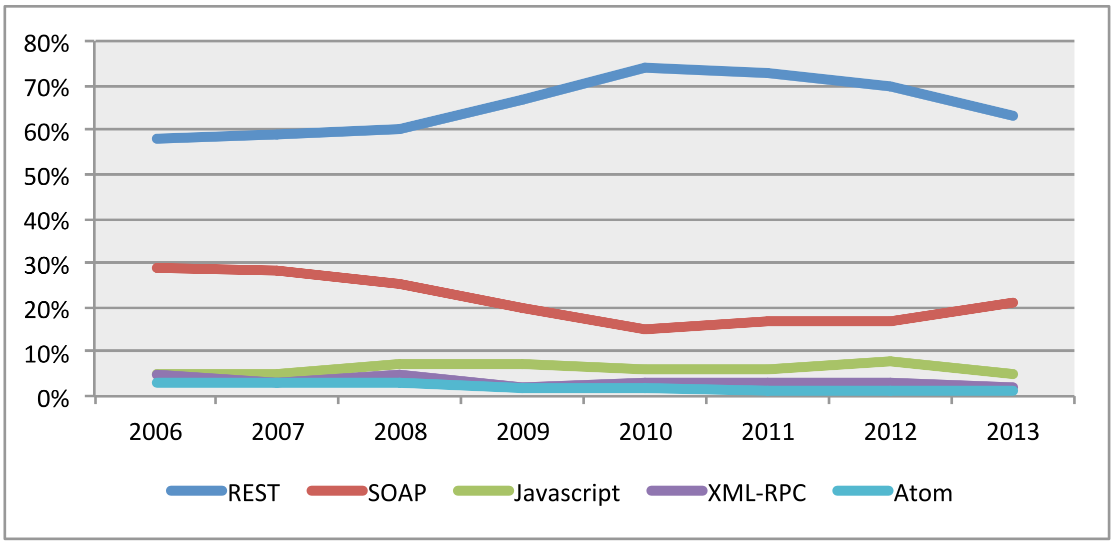

FIGURE 1.1: Service protocol landscape - source: programmableweb.com

business domains. The information shown in Figure 1.1 does not necessarily aim to prove to the reader that Representational state transfer (REST) is a better than SOAP, but rather to show that REST is more popular and that it is widely used in open APIs. In addition, it shows that there are more protocols and styles available in the state of the art for developing re-useable and remote services.

From a different angle, the aspect of creating opportunistic combinations of services was solved through Mashup editors that have much better support for RESTREST services but at the same time, do not offer a type system and as such they are not aiming towards production grade quality. A common characteristic that's missing from both of the aforementioned approaches is the support for session. By session we mean a single and identifiable instantiation of a runtime instantiation that can be used for the purposes of maintaining the state of a system from a user's perspective. This aspect is very interesting when visiting the problem of service composition from a telecommunications perspective.

Resuming our protocol investigation, an aspect that is concealed by the previous figure is that of the protocol ecosystem used within the telecommunication domain as a result of the transition to Third Generation (3G) [43] packet switched networks. This transition, allows the merging of the two most successful paradigms in communications: mobile communications and the Internet. One of the key elements in this transition is the IP Multimedia Subsystem (IMS) [25]. The IMS is an architectural framework for delivering Internet protocol (IP) multimedia to mobile users. Besides mechanisms for QoS and session management, one of the main goals of IMS is the provision of integrated services. More specifically, it is the establishment of an ecosystem, where it is deemed possible for an operator, to integrate native services with third parties. The benefit of such an establishment is the minimization of Time-To-Market [153] and the insurance of robustness, through the reuse of stable constituent services.

Convergence between the traditionally separated IT/internet, enterprise and telecommunication industries is ongoing and gains significant momentum from current and future market demands. The Internet of things [11], the expectation of more than 50 billion connected devices [51] in a few years and the ever-increasing popularity of smart-phones are only a few examples. They mark an inflationary growth of assets with increased and at the same time highly diverse communication demands. In order to accommodate this growth, networks will not only connect devices to each other, but they will also provide a dynamic service infrastructure as backend. The future service landscape, as outlined here, is diverse in using very different technologies to implement services. The ability to compose heterogeneous services is essential and central for such future networks.

In conclusion, we are witnessing the merging of the Internet and mobile communications. These domains have large audiences and provide a plethora of services with distinct characteristics. This merging, along with the trend of Web 2.0 and mashups, results not only into an extraordinary increase in the amount of available online services, but also to a need for separating information and presentation in ways that allow for novel forms of reuse. However, the large amount of services and the increased complexity between service interactions make the act of browsing, combining and reusing online services a great ordeal.

1.2 Motivation

The following issues that have been identified in the state-of-the art motivate our approach to the challenge of applying service composition, in heterogeneous domains.

The first issue is identified in workflow languages and frameworks such as the ones used in WS-BPEL. Such frameworks are limited to interacting only with external services that are either directly or indirectly exposed as Web Services following the Web Services Definition Language (WSDL) standard. This limitation is rather crucial to the telecommunications domain, where services are usually made available via several other protocols such as Session Initiation Protocol (SIP).

The second issue is that application development in those frameworks is done using static type systems. This means that at design-time, the development environment would mark a workflow as invalid if a type error occurs and at runtime the framework would abort the execution. This is an issue because at design-time, it can very well be the case that the workflow developer is not aware of the exact types of the variables, the workflow is going to be using at runtime.

From a parallelism and concurrency point of view, in a workflow one needs to define execution of parallel tasks explicitly using specific constructs. This increases the amount of effort for the workflow designer because she needs to use additional constructs and implement additional logic in order to make the workflow capable of utilizing multiple cores, which are standard in most computer systems these days. In addition, cognitive research has shown that the addition of such constructs increases the complexity of the workflow and makes it difficult to understand and maintain.

These experiences motivate a strong demand for a language that is able to specify compositions in heterogeneous service environments and that is still relatively easy to understand and use. This thesis discusses which properties a language for specifying this type of service compositions should have. The resulting language aspires to be designed in a way that results in a significant progress to the convergence of industries and technological domains.

1.3 Contributions

The contributions made by this thesis can be grouped into the design of a concurrent-by-default graphical language, which we decided to name SCaLE, that allows for describing service compositions of heterogeneous services and a fine-grained, technology agnostic composition framework responsible for interpreting the constructs of the proposed language and interacting with external services by means of execution agents.

A major contribution of this thesis is Service Composition LanguagE (SCaLE) — a graphical language for heterogeneous service composition. SCaLE is deprived of constructs for parallel execution of actions. Instead, all actions within SCaLE are executed concurrently by default. Moreover, it supports dynamic service selection of services at run time. Compositions in SCaLE are comprised of atomic and composite actions. Atomic actions are responsible for executing one function and afterwards provide the result of the computation of that function. Composite actions are actions that can contain nested composite or atomic actions. In SCaLE the execution flow of a service is defined using data dependencies and events. Data dependencies connect data outputs and data inputs of actions. Events can originate from the underlying composition framework. To avoid possible race conditions, only copies of variables are being transferred between actions in order to avoid changing the original ones. The original value of a variable can only be changed by special variables known as effects.

Due to the fact that SCaLE has a graphical notation it is easier for designers with non-IT workflow background to learn and comprehend the language and build compositions. In addition, SCaLE has formally specified execution semantics. Even though it is possible to implement service compositions that may lead to deadlocks, it is possible to detect them in advance due to the graph like nature of the language. We evaluate the proposed language qualitatively in terms of workflow patterns in order to measure its expressiveness as opposed to its closest rivals.

Concerning the composition framework, this thesis describes an asynchronous, event-driven non-blocking composition framework that possesses the following characteristics:

- Integration of heterogeneous service technologies within a single composite application
- Optional type system
- Service interaction
- Full control of execution flow through the use of composite application skeletons described using SCaLE

Within the premises of the composition framework, the problem of dealing with heterogeneous services is dealt by the convention of technologic specific execution agents, thereby permitting the core to be technology agnostic. The proposed framework is evaluated in terms of overhead and throughput.

1.4 Constituent Papers

Part of the research presented in this thesis has been published in the following internationally peer-reviewed publications.

- J. Niemöller, E. Freiter, **K. Vandikas**, R. Quinet, R. Levenshteyn, I. Fikouras: **Composition in Heterogeneous Service Networks: Requirements and Solutions**, 2012/1/1, Business System Management and Engineering, Springer, Berlin, Heidelberg
- **K. Vandikas**, R. Quinet, R. Levenshteyn, J. Niemöller: **Scalable service composition execution by means of an asynchronous paradigm**, Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference
- **K. Vandikas,** N. Liebau, M. Döhring, L. Mokrushin, I. Fikouras: **M2M Service Enablement for the enterprise**, Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference
- J. Niemöller, **K. Vandikas**, R. Levenshteyn, D. Schleicher, F. Leymann: **Towards a service composition language for heterogeneous service environments**, Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference
- **IMS Application Developer's Handbook, Elsevier**, Contributed a chapter on SIP and AJAX interoperability, 2011
- J. Niemöller, **K. Vandikas**: **SCALE — A language for dynamic composition of heterogeneous services**, 15 December 2010

- **K. Vandikas**, E. Freiter, R. Levenshteyn, R. Quinet, J. Niemöller, I. Fikouras: **Blending the telecommunication domain with Web 2.0 services**, Intelligence in Next Generation Networks (ICIN) 2010 14th International Conference **(Best presentation Award)**
- J. Niemöller, E. Freiter, **K. Vandikas**, R. Quinet, R. Levenshteyn, I. Fikouras: **Composition in Converged Service Networks: Requirements and Solutions**, International Workshop on Business System Management and Engineering (BSME), 2010
- J. Niemöller, E. Freiter, **K. Vandikas**, R. Quinet, R. Levenshteyn, I. Fikouras:**Multi-Technology Service Composition for the Telecommunication Domain - Concepts and Experiences**, Next Generation Mobile Applications, Services and Technologies (NGMAST) 2010 **(Best paper nominee)**
- **K. Vandikas**, J. Niemöller, I. Fikouras: **Service creation in the long-tail marketplace**, Symposium Innovating IT, March 5, 2009

1.5 Thesis Outline

The second chapter, **"Background and Related Work"** examines the state of the art for existing approaches to service composition. The examination starts by presenting the different techniques which have been used in the telecommunications domain and thereafter moves to review approaches used for Web services composition in the IT domain. The complete set of approaches for service composition is assorted in three categories: Static, Artificial Intelligence Planning and Domain-specific language based.

The third chapter, entitled **"The approach"** presents a set of requirements and challenges found in the academic state of the art, but also in the industrial. By assessing the extent to which the approaches mentioned in chapter, address said requirements we move forward to detail the guiding key concepts of our proposed approach to a composition framework and a language for defining service compositions.

The fourth chapter, **"System Design"**, moves from an abstract description of our proposed approach to a concrete realization. This chapter begins by detailing the architecture of the proposed framework for service composition and its constituent components. Afterwards it provides a complete description of the proposed language for composition, named SCaLE and the different constructs that constitute that language, along with a few examples of compositions written in SCaLE. To complement the description of SCaLE, in Appendix B, we provide the description of formal semantics for the proposed language.

The fifth chapter, **"Implementation"** contains the implementation details of the system designed in the previous chapter. More specifically, we describe the inner-workings of the proposed composition framework and its constituent components. This chapter concludes by evaluating the quality of the source code base that has been created.

The sixth chapter, **"Evaluation"** provides an evaluation of the proposed system. First we begin by evaluating the proposed composition framework in terms of response-time and throughput as opposed to its closest rival framework. Then we proceed by doing a qualitative comparison between SCaLE, our proposed language for service composition and two widely used and standardized languages used in the state of the art, BPMN 2.0 and WS-BPEL. A more detailed description of the qualitative analysis is given in Appendix A.

The seventh chapter, **"Conclusions"**, summarizes the set of contributions that have been made by this thesis.

The eighth chapter, **"Future work"**, contains a brief list of new research directions that can be made in the area of service composition, in association with the proposed composition framework and language.

In Appendix A, we provide the specifics of the qualitative comparison that we have made between our proposed language and state of the art languages. Appendix B details the formal semantics for the proposed language is documented.

At the end of this thesis we provide a list of abbreviations to assist the reader with the different terms that we have used throughout this document.

Chapter 2

Background and related work

The process of service composition is very similar to the process of writing any software. It is consisted of two phases, a design-time and a runtime. At design-time, the designer (or programmer) is making a plan of the different steps that are needed to solve a particular problem; From these steps, later on, the designer derives the different functions that will be used in order to solve the problem and the flow in which said functions should be executed to solve this problem. Later on at design-time a virtual machine is used in order to convert the expression, or the domain specific language used by the designer to a set of constructs which are either directly, or indirectly [39] associated with the instruction set of the host environment. The main difference between the process of service composition and the process of writing any software is that service composition comes by default with a rather large set of pre-built functionality which is hosted remotely, as opposed to most programming languages which come by default with a large set of libraries which are used to operate with the host environment (i.e. I/O functions). This property enables the designer to focus on writing queries instead of function calls when selecting external functionality and as such permits the designer to focus on the flow in which such functions are going to be executed.

This chapter explores the area of service composition by examining the work that has been done until now. The chapter begins by describing techniques, which are used in the telecommunications industry in order to perform a type of service composition that is done manually by the developer and is closely related to the properties of the telecommunications network. The list of those techniques is later on complemented with an additional set of approaches used for web service composition. Overall, research in the area of service composition can be divided in three different categories:

1. Static approaches
2. Artificial Intelligence Planning
3. Domain-specific language based

The aforementioned categorization generalizes on the Web service composition classification as previously identified and reviewed by Dustdar et al. [47] and Rao et al. [133].

Our approach is geared towards producing a design methodology and a system that is on the one hand-side suitable to the demands of telecommunication networks with regards to availability and performance and on the other hand-side allows the user to easily modify, extend and control the composition.

2.1 Definitions

This section contains a brief list of definitions that are going to be used in this chapter and throughout the remaining chapters of this thesis.

Service: A service [84] is a logical representation of a repeatable activity that has a specified outcome. It is self-contained and as such, it is perceived as a 'black box' to its consumers.

SOAP: SOAP [71] is a protocol specification for exchanging structured information in the implementation of Web services in computer networks.

WSDL: WSDL [35] is a machine-readable Intermediate Description Language (IDL) used for describing the functionality that is offered by a web service. As such it is limited to describing the syntax of messages that enter or leave a computer program.

Web service: According to World Wide Web Consortium (W3C) [74], a web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine processable format (specifically WSDL). Other systems interact with a Web service in a manner prescribed by its description using SOAP message, typically conveyed using Hypertext Transfer Protocol (HTTP) with an Extensible Markup Language (XML) serialization in conjunction to other Web related standards.

Workflow: A workflow [165] consists of a sequence of connected steps. It is a graphical representation of a sequence of operations declared as the work of a person or a group thereof, an organization or one/more simple or complex mechanisms.

Composition: A collection of activities implementing a business goal [120]. The execution order of these activities may change depending on a variety of influencing factors such as the state of the surrounding infrastructure. The term composition is used instead of the term workflow when heterogeneous services are being composed rather than placed in a certain order for execution.

Heterogeneous Composition: Composition that utilize constituent services from various technological domains within a single composition [120].

Service composition is a design principle, applied within the service-orientation design paradigm, which encourages the design of services that can be reused in multiple solutions that are themselves, made up of composed services [52]. The ability of the service to be recomposed is ideally independent of the size and complexity of the service composition. As such, service composition is defined as the process of combining and linking existing services (atomic or composite) to create new working services.

SOA: In software engineering, a SOA [171] is a set of principles and methodologies used for designing and developing software in the form of interoperable services. Each service encapsulates well-defined business functionality and it is built as a reusable component. Thereafter, following the principle of service composition, new services can be generated as a coordinated aggregate of pre-existing services.

2.2 Service Composition in the telecommunications industry

There are a number of approaches to static service composition in the telecommunications industry [19]. However, only three approaches have been standardized for use over Internet Protocol (IP) networks: Third Generation Partnership Project (3GPP) IMS [25], Java APIs for Integrated Networks Service Logic Execution Environment (JSLEE) [55] and the SIP Servlet API (JSR116 [94], JSR289 [175]). In our review, we will focus on the various techniques that are used for service composition within IMS and in particular with the SIP Servlet API, as JSLEE was not widely adopted by the industry [34]. In addition, we will also cover Distributed Feature Composition (DFC) [86] due to its role in influencing JSR116. The main premise of DFC is the lack of a global state in service interaction. Based on this premise, DFC allows for highly efficient mechanisms for the creation of simple compositions, in a pipes-and-filter architecture. On the other hand-side IMS is more liberal with regards to service composition, either by outsourcing this task to JavaEE [87] based application servers that can host any kind of imperatively described systems, by using linear and static associations of services and last but not least by using rule engines.

2.2.1 Distributed feature composition

Jackson et al. [86] coined the term DFC, which is a virtual architecture for telecommunication systems and more specifically connection services. A connection service is defined as a distributed software system that makes and manages dynamic network connections. A connection service enhances basic network protocols to offer added value to the user. Connection services are organized and built in terms of features in order to ensure flexibility. A feature is defined as an "increment of functionality with a coherent purpose" and corresponds to a component type. Components communicate through internal calls. Describing its corresponding component type and the rules for including the component instances into configurations specifies a new feature.

Each call is handled by dynamically assembling a configuration of instances of these components, according to the features to be applied to that particular call. The

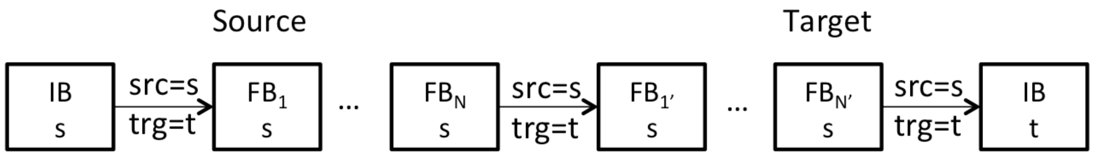

FIGURE 2.1: A simple DFC usage with two interfaces boxes (IB) and N feature boxes (FB) in each part of the network

resulting configuration is analogous to an assembly of pipes and filters were feature components are independent, they do not share state, they do not know or depend on which other feature components are at the other ends of their calls (pipes), they behave compositionally, and the set of them is easily enhanced. This relationship is illustrated in Figure 2.1. IB stands for Interface Box and it is used to denote a piece of functionality dedicated to providing an interface to FB which stands for feature box. Feature box contain the implementation of specific features such as 'call forwarding on busy.' Source and target are used to represent different networks (perhaps operated by different vendors) that host different features and different interfaces to those features.

The composition is implemented by a dynamically assembled graph of feature boxes (nodes) and calls (edges). This graph is called a usage. DFC thus leads to a pipes-and-filters architecture in which calls are the pipes and feature boxes are the filters. A source feature applies to any connection request with the appropriate source address. A target feature applies to any connection request with the appropriate target address. This provides an important mechanism for customization of services. Interoperation features provide conversion functions implemented as gateways to other domains.

One of the fundamental assumptions behind DFC is that features should be transparent. When their functions are not required they should act transparently and not influence the connection in any way. The pipes-and-filters arrangement gives features autonomy, when their functionality is required they can interrupt the connection and carry out their function without external intervention.

The DFC approach clearly has had considerable influence on state of the art technologies such as JSR116 that is described later on in paragraph 2.2.2.2.

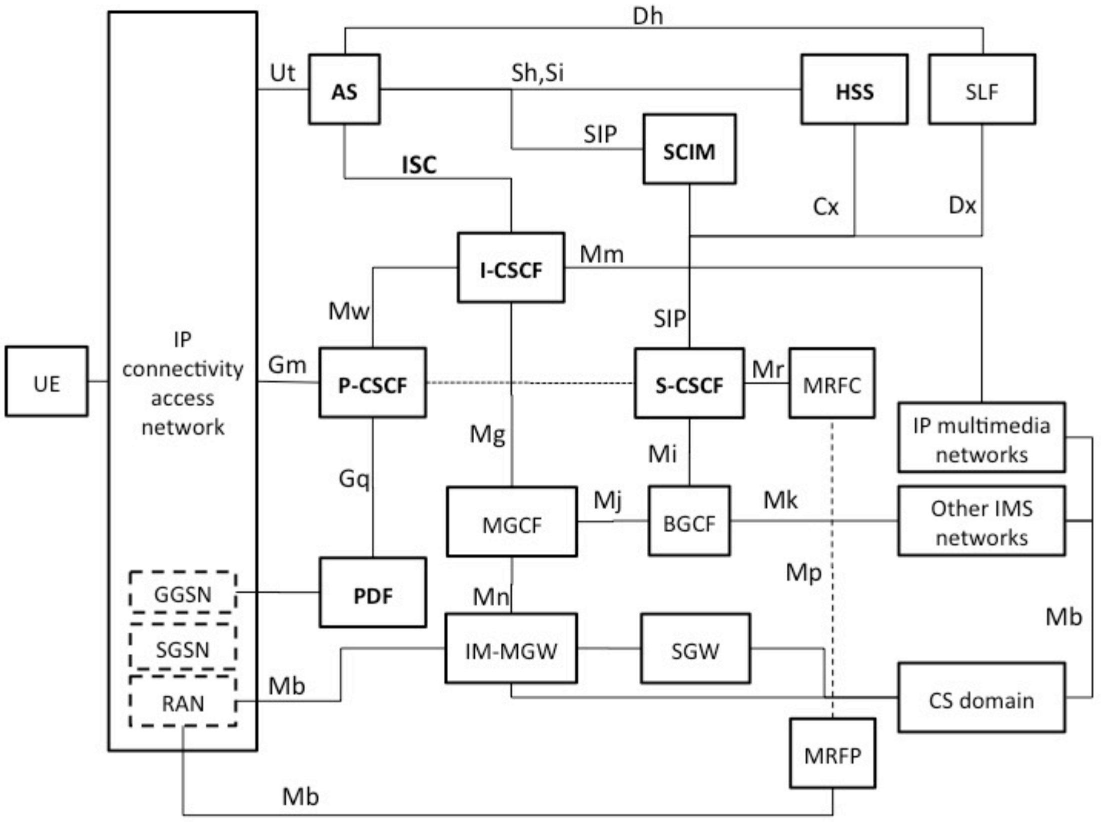

FIGURE 2.2: TISPAN IMS Architecture with Interfaces

2.2.2 IP Multimedia subsystem

In the telecommunication industry, a transition to native support for IP packet transportation has taken place through the migration to Third Generation (3G) packet-switched networks. IMS is the Next Generation Networking (NGN) architecture for telecommunications services, standardized by 3GPP and the European Telecommunications Standards Institute (ETSI) Telecoms & Internet converged Services & Protocols for Advanced Network (TISPAN) group [25]. IMS was defined using Internet protocols standardized by the Internet Engineering Task Force (IETF). Specifically, IMS builds on SIP [143] with a few extensions standardized by 3GPP and running over IP. It is an architectural framework for delivering IP multimedia services. Figure 2.2 illustrates the functional entities and reference points which are used in IMS.

One of the most important features of IMS, with regards to service composition, is that it allows for a SIP application to be dynamically and differentially triggered based on a user's profile. This feature is implemented as a filter-and-redirect signaling mechanism in the Serving Call Session Control Function (S-CSCF), which is analyzed in paragraph 2.2.2.4. The S-CSCF may apply filter criteria to determine the need to forward SIP requests to an Application Server (AS). At this point it is important to point the distinction between the originating network; the network that hosts the originating party and the terminating network that contains the terminating party. With this distinction in mind service composition is performed in its corresponding network.

Camarillo et al. [25] and Bertrand [16] provide a complete description of each entity and reference point depicted in Figure 2.2. For the purposes of this thesis, we will focus primarly on:

- Home Subscriber Server (HSS)
- Call Session Control Function (CSCF)
- AS
- Initial Filter Criteria (iFC)
- Intelligent Network (IN)
- Charging
- Policy charging and rules functions (PCRF)
- Service Capability Interaction Manager (SCIM)

Since these nodes are the most relevant to the process of service composition which we are addressing.

2.2.2.1 Home Subscriber Server

HSS is a master user database that supports the IMS network entities that actually handle calls. It stores subscription-related information (subscriber profiles), performs authentication and authorization of the user and can provide information about the subscriber's location and IP information.

2.2.2.2 Application Servers

SIP application servers (AS) are used for hosting and executing services. Depending on the actual service, the AS can either function in a SIP proxy mode, SIP User Agent mode (UA) or a SIP Back-2-Back user agent mode (B2BUA).

An AS can either be located in the home network or in an external third party network. If it is located in the home network, it can query the local HSS using the Diameter Sh or Si interfaces. Java specification requests (JSR) such as the JSR116 and JSR289 standardize the functionality of SIP application servers. The following paragraphs, JSR116 and JSR289 provide a short descriptions of what is standardized by the aforementioned JSRs.

JSR116 JSR116 is a SIP Servlet Specification. It standardizes the functionality of SIP Containers. Since it is a JSR, it describes Java specific APIs for SIP Servlets. One of the goals of this specification was to provide an approach similar to the well-known HTTP Servlets approach (JSP), so that developers can easily learn and start using it. This basic approach introduces a SIP Container, where multiple SIP applications consisting of SIP services can be deployed. Since it is particularly well suited for server-side SIP applications, this specification is usually taken as a basis for most SIP Application Servers (e.g. SailFin [150], BEA WLSS [66], etc.).

This specification describes different aspects of SIP Servlets development, as well as deployment and execution of the servlets on the SIP container.

For the development phase, JSR116 defines a set of SIP servlet APIs, which provide a framework for the SIP servlets and formalize the method how such servlets access the SIP functionality, react to incoming/outgoing SIP packets and interact with the SIP container.

For the deployment phase, the specification introduces the deployment descriptor format (e.g. SIP.xml file format), which is used to describe the set of SIP applications and servlets provided by a given deployment package. In addition, it defines a set of rules that specify the conditions when the SIP applications and servlets deployed in a given package should be invoked and the order of such invocation. It is permitted and possible that the same SIP Container has several applications and servlets that can be invoked on the same incoming SIP packet, according to the rules described in their deployment descriptor. Thus, rather limited composition functionality is provided. It allows for static composition of linear sequential chains of SIP servlets invocations.

Moreover, JSR116 describes the behavior of a SIP container at runtime. It specifies how the container towards the applications and servlets responsible for their processing propagates SIP requests. Moreover, it specifies the management of runtime dependencies between the SIP applications and servlets belonging to them. The task of managing all this runtime relationships and dispatching of SIP messages is handled by the SIP Servlet Dispatcher.

In an effort to overcome some of the problems discovered and experienced after the introduction of JSR1116, an update of the specification was proposed in form of a JSR289, which will be adressed in the next sections.

JSR289 JSR289 is an attempt to extend the SIP servlets specification. Extensions take into account experiences gained with JSR116. One of the very popular features asked by many participants of JSR and practitioners is an improved support for composition of SIP services. There are requests to make it more dynamically configured, to clarify some parts of JSR116 and make them more precise, and to improve the possibility to interact with other technologies (e.g. Enterprise Java Bean (EJB)s, Web Services, HTTP Servlets) deployed on the same application server.

2.2.2.3 BEA WebLogic SIP Server

A SIP servlet container, according to the JSR116 standard, hosts servlets accessible via SIP. Moreover, it provides the means of creating chains of servlets. This is meant to allow the composition of more complex services as chains out of individual servlets. Consequently, the SIP container acts for the SIP services it hosts as a composition execution engine. Currently, most SIP servlet containers implementing JSR116 use a static composition approach. The equivalent to composition creation takes place manually at deployment time and is static in the way that it cannot be changed without a change in the configuration of the container.

Advanced solutions are slightly more dynamic. BEA WebLogic SIP Server (WLSS) tries to move the split between composition creation and composition execution

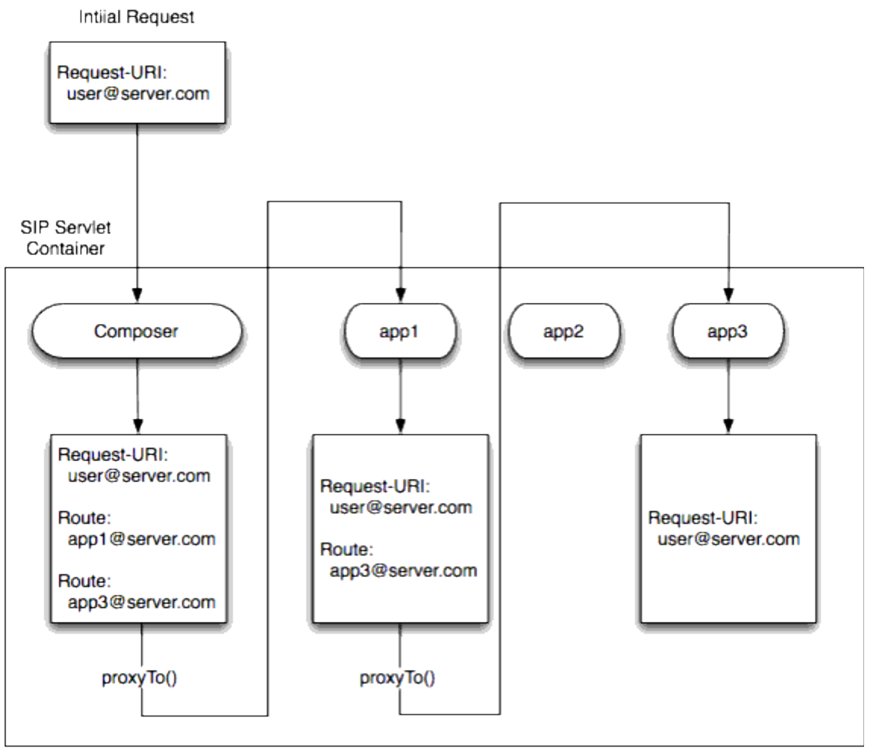

FIGURE 2.3: BEA WLSS SIP Servlets composition

towards the runtime phase by introducing a special composer servlet. The composer servlet always acts as the first component of the composition. This servlet effectively performs service creation based on algorithms hard-coded during its design and possibly by consulting external nodes or databases. However, changes to the created chain of SIP services are no longer possible after composition execution starts.

2.2.2.4 Call Session Control Function (CSCF)

An IMS core consists of three different CSCFs:

- Proxy-CSCF (P-CSCF)
- Serving-CSCF (S-CSCF)
- Interrogating-CSCF (I-CSCF)

These three entities are based on SIP application servers and are used to process SIP signaling packets in IMS. A CSCF is similar to a Mobile Switching Center (MSC) in the Global System for Mobile Commnunications (GSM) domain, with the additional capability of supporting multimedia sessions along side voice calls. Moreover, CSCFs can generate Call Detail Records (CDR)s for charging and billing purposes.

Proxy-PCSF The P-CSCF is the first point of contact in an IMS terminal. As such, it is assigned to an IMS terminal before registration and does not change for the duration of the registration. It is positioned in the signaling path and can inspect every signal. It provides subscriber authentication and it can be used to prevent spoofing and replay attacks in order to protect the privacy of the subscriber. Moreover, it supports compression, thus reducing the round-trip over slow radio links. Finally yet importantly, it may include a Policy Decision Function (PDF) that authorizes media plane resources such as Quality of Service (QoS) over the media plane.

Serving CSCF The S-CSCF is located in the home network and its purpose is to perform session control and registration services for physical devices or terminals (user endpoints). As such, it handles SIP registrations, which allows it to bind the user location (e.g., the IP address of the terminal) and the SIP address. It sits on the path of all signaling messages of the locally registered users, and can inspect every message. It decides to which application server(s) the SIP message will be forwarded, in order to provide their services. Finally, it provides routing services, typically using Electronic Numbering (ENUM) lookups. It enforces the policy of the network operator.

There can be multiple S-CSCFs in a network for the purposes load distribution and high availability. It's the HSS that assigns the S-CSCFto a user, when it's queried by the I-CSCF. There are multiple options for this purpose, including mandatory/optional capabilities to be matched between subscribers and S-CSCFs.

Interrogating-CSCF Interrogating Call Session Control Function (I-CSCF) is used as a contact point for all connection destined to a user related to this network operator, or a roaming user within the operators service domain. It queries the (HSS) to retrieve the address of the S-CSCF and assign it to a user performing a SIP registration. Moreover, it forwards SIP requests or responses to the S-CSCF.

2.2.2.5 Initial Filter Criteria (iFC)

From a SOA perspective, IMS uses a simple orchestration mechanism in order to select the additional services that are needed within a telecommunication session [44]. This mechanism relies on iFC. iFC are filter criteria that are stored in the HSS as part of the IMS Subscription Profile and are downloaded to the S-CSCF upon user registration (for registered users) or on processing demand (for services, acting as unregistered users). They represent a provisioned subscription of a user to an application. iFCs are valid throughout the registration lifetime or until the User Profile is changed. The term Shared iFC denotes an iFC, which, due to its common use for a large number of subscribers, is only referenced in the Subscription Profile and provisioned on a different path between the HSS and the S-CSCF.

An iFC is composed of:

- An Application Server Uniform Resource Identifier (URI) where the request is to be forwarded in case of a match.
- A Trigger Point in the form of a logical condition that is verified against initial dialog creating SIP requests or stand-alone SIP requests.

As such, iFCs are specific to a user and represent a list of services to be invoked. This method is not dynamic; the services that a user might need are placed in the chain of services regardless of their usefulness for a given session. Service interaction is managed in the most basic way by manually defining service combinations and iFCs so that it works without causing any interaction problems.

2.2.2.6 Intelligent Networks (IN)

The aim of IN was to enhance core telephony services offered by traditional telecommunications networks, which usually amounted to making and receiving voice calls, sometimes with call divert. This would then provide a way for operators to build services in addition to those already present on a standard telephone exchange. Later a new variant of IN standard evolved. This variant was called Customized Applications for Mobile networks Enhanced Logic, or CAMEL for short.

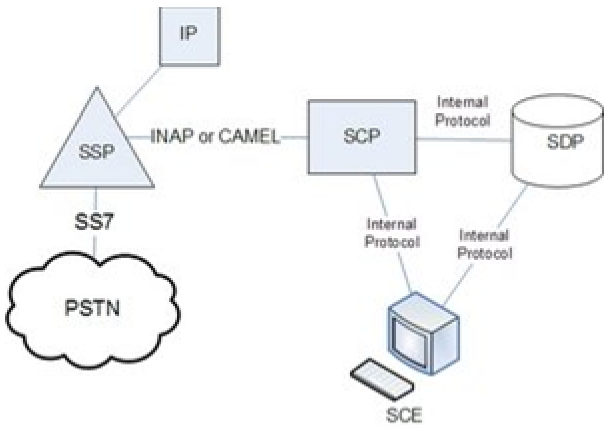

Figure 2.4: IN architectural components

This allowed for extensions to be made to the mobile environment, and allowed mobile phone operators to offer the same IN services to subscribers whilst they are roaming as they receive in the home network.

IN represents a nice example of a static service composition with a clear separation of the service creation and service execution phases.

Service Switching Function (SSF) or Service Switching Point (SSP) is co-located with the telephone exchange itself, and acts as the trigger point for further services to be invoked during a call. In this sense, it plays the role of a dispatcher.

The SSP implements the Basic Call State Machine (BCSM) that is a Finite State Machine (FSM) that represents an abstract view of a call from beginning to end (off hook; dialing; answer; no answer; busy; hang up etc.). Consequently, the SSF maintains the overall state that allows it to make decisions regarding the execution of the service.

For each state (and corresponding Detection Points) the SSP can signal the Service Control Point for further instructions on how to proceed. This is called a trigger. Trigger criteria are freely defined based on related data such as the subscriber calling number or the dialed number.

The Service Control Function (SCF) or Service Control Point (SCP) is responsible for the execution of compositions. It contains service logic which implements the behavior desired by the operator as a form of static service composition. During service logic processing, additional data required to process the call may be obtained from the Service Data Function (SDF). The logic on the SCP is created using a Service Creation Environment (SCE).

The SCE is the development environment used to create the services on the SCP. Although any type of environment can be used for development, low-level languages like C are rarely used. Proprietary graphical languages are used often to enable telecom engineers to create services directly. These services represent essentially static service compositions, which cannot be modified or extended during execution.

SDF or Service Data Point (SDP) is described in subsection 2.2.2.7.

Specialized Resource Function (SRF) or Intelligent Peripheral (IP) is a node that can connect to both the SSP and the SCP and delivers additional special resources into the call. Examples of such resources are play voice announcements or collect Dual Tone Multi-frequency Signalling (DTMF) tones from the user.

2.2.2.7 Charging System

A charging system is consisted of two nodes that help compose appropriate charging for the requested services. The Charging Control Node (CCN) enables charging for General Packet Radio Service (GPRS), Short Message Service (SMS) and content-based services. This system can be used as a stand-alone node or as part of the overall Charging System. The SDP on the other hand is responsible for rating calls and events, post processing of CDR and initiation of Unstructured Supplementary Service Data (USSD) notifications. Depending on the service type, the CCN acts as a relay between the core network MSC, Gateway GPRS Support Node (GGSN) and the Serving GPRS Support Node (SGSN), service and Multi Mediation network (e.g. MMC), and SDP that is the part of the Charging System. This element supports the following services in the Charging System:

- Online charging of GPRS and SMS using Camel Application Part Version 3 (CAPv3) API
- Online charging of Content and Event, using the Diameter protocol [24]
- The Policy and Rating Server (PRS) using the Diameter protocol
- Relay of Charging Based Service Control Signalling System Version 7 (SS7)

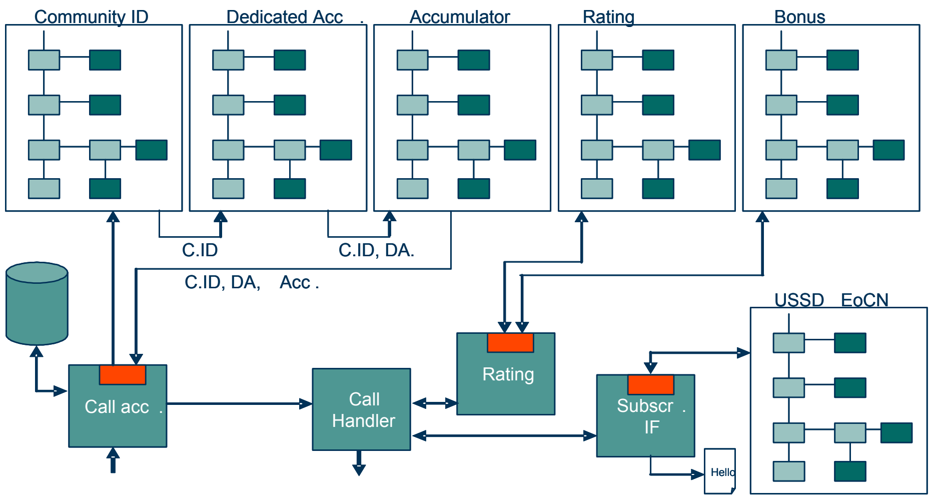

FIGURE 2.5: SDP-PSC Selection Tree Component

The CCN node utilizes different charging and control rules depending on the service type, e.g. online charging of GRPS traffic either is done according to connection duration or transferred volume. A configurable conversion factor allows performing conversions between time and volume. CCN can be parameterized to provide different functionalities depending on the operators' needs. This feature characterizes other Charging System nodes and allows operators to deliver new market offers even without additional implementation efforts.

The charging SDP on the other hand takes decisions regarding which tariff to use in a particular situation dynamically during runtime. This is achieved by a selection mechanism that uses the data stored in selection trees, as well as data on the subscription of the user and the requested service to ”compose” the appropriate tariff. Therefore, this approach can be considered as a specialized form of composition.

SDP holds or administers: subscriber data, account data, service class data, announcement class data, usage accumulators data, tariff and charging analysis data, subscriber life-cycle data, dedicated accounts data, license management data, USSD and SMS notifications, promotion plan data, Home Location Register (HLR) blocking data and community data.

These trees are managed by a dedicated service component that is used for taking decisions based on the trees it manages for many of the functions of the SDP. This decision support function implements both tree management functionality to

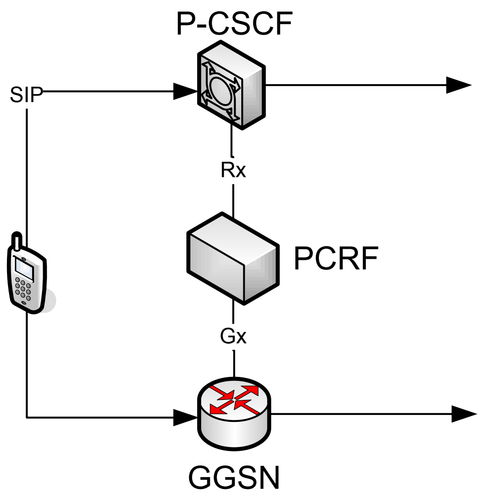

Figure 2.6: 3GPP standardized view of PCRF deployment

edit and test data, e.g. via Tariff Management, as well as flexible mechanisms to select/calculate and return data.

2.2.2.8 Policy charging and rules function (PCRF)

PCRF function is a software node designated in real-time to determine policy rules in a multimedia network. EPC is Ericsson's implementation of the 3GPP PCRF node. The EPC is developed and tested in a platform agnostic environment using POSIX standards. EPC 1.0 and 2.0 are integrated in Ericsson's proprietary operating system Dicos [54].

EPC 1.0 interacts with the Service Aware Support Node (SASN) using the SRAP/Diameter draft8 interface. Later versions that will integrate SASN with the GGSN will handle this communication over the Gx interface. EPC 2.0 interacts with external entities using Rx & Gx Diameter based interfaces and old EPC 1.0 for migration support. Both EPC 1.0 and 2.0 are integrated with the P-CSCF and collapse the Rx interface. Rx support was added in 2007 to support separate P-CSCF and other entities supporting the Rx interface (e.g. streaming server). EPC can be deployed as shown in Figure 2.6.

The EPC contains a rule engine used to evaluate rules and policies towards making decisions about which policy to apply in a specific situation. Policies and objects are stored in the Dicos built-in database (DBN) and can be manipulated through Lightweight Directory Access Protocol (LDAP) interfaces.

EPC policies are comprised of rules and described in a proprietary language, which is a subset of Ecmascript [48]/Javascript. Rules can use objects in DBN and objects defined in external LDAP. The internal DBN cannot be expanded at runtime; therefore, new objects cannot be added at runtime. However, this can be achieved using external LDAP repositories. Conflicting results can be resolved using various conflict resolution algorithms. In addition, conflict resolution can be applied between policies (which algorithm to use is decided by the policies themselves).

The EPC does not impose a selection mechanism for data, which makes the rule engine flexible and expandable. The data to select which policies to use is arranged between the requestor and a policy locator.

2.2.2.9 Service Capability Interaction Manager (SCIM)

SCIM was introduced in 3GPP TS 23.002, as a function within the SIP Application Server domain of the IMS architecture. The role of the SCIM is that of a service broker in more complex service interaction scenarios than can be supported through a service filtering mechanism. As an example, feature interaction management provides intricate intelligence, such as the ability to blend Presence and other network information, or more complex and dynamic application sequencing scenarios.

The SCIM as proposed to the 3GPP uses IP Multimedia Service Control (ISC) and SIP interfaces to enable composite service behavior leveraging simpler service components. Consequently, service composition functionality as encountered in a SOA is well suited for implementing the SCIM function.

One could therefore envisage the SCIM to be a programmable engine, that implements service composition functionality enabling the creation and execution of composite services. Typically, such engines are implemented based on a data-driven approach i.e. rules-based or model-driven approach.

The term SCIM actually refers to an entire class of functions and not to a single node that handles this broad problem. Consequently, one can assume that SCIM functionality can be implemented in a number of different components each addressing different requirements.

Different SCIM types can be broadly classified into the following three categories:

- **AS Internal Function**. A SCIM implemented as a request dispatcher within the local execution environment. This would be similar to function of the dispatcher in a JSR116 SIP servlet container. More SCIM like functionality would therefore fall within the scope of the dispatcher in JSR289.
- **Format Controller**. A SCIM that is developed and optimized for supporting one particular communications service. Such a SCIM could be embedded in the implementation of a service format.
- **General Purpose SCIM**. This type of SCIM manages interaction between components that implement SIP proxies or user agents, service capabilities that are exposed using WSDL and SOAP-based abstractions of the IMS network or SIP features and legacy signaling system components.

2.3 Web service composition

Web service composition consists of a collection of approaches that either automatically generate a workflow based on a given problem description, or require a manual definition of a workflow from a designer, and thereafter complement the service composition process at runtime. In this section, we divide web service composition in three categories:

- Static
- Artificial Intelligence Planning
- Domain-specific language based

The following subsections (2.3.2, 2.3.3, 2.3.4) focus on describing the main characteristics of each category. Prior to these descriptions, we provide a brief intro to workflow languages such as Web Services Business Process Execution Language

(WS-BPEL) and Business Process Model and Notation (BPMN) 2.0. This input is important for two reasons:

1. These languages are standardized, they are widely used and they posses formal descriptions describing the semantics of their execution
2. The form a common denominator for most composition approaches, since the result that is produced by the aforementioned methods is usually expressed as a workflow described in these languages and is later on sent on to be executed at runtime by the corresponding execution framework.

2.3.1 Workflow languages

2.3.1.1 WS-BPEL

WS-BPEL [42] or BPEL for short is the de-facto standard to describe workflows in Web service environments. A BPEL process is built using basic and structural activities. Basic activities are responsible for the communication with the environment or the transformation of data. An invoke activity, for example, is used to invoke other Web services or BPEL processes. Another example would be an assign activity that is used to copy data within a BPEL process. An example for a structural activity is the while-loop activity.

BPEL processes are executed on top of the Web services layer. They use Web services for calculations and orchestrate them to get a more complex solution. Due to the tight coupling WS-BPEL has with Web Services, it is not applicable to heterogeneous service environments directly. However, indirectly, one can always implement a Web service based adapter on top of the original technology used for a service and thus make it accessible to BPEL. BPEL as such lacks a graphical representation. Thus, for business process designers with no IT background the learning curve may be steeper that with other languages. BPEL has been standardized by the Organization for the Advancement of Structured Information Standards (OASIS) and is widely used in the industry.

2.3.1.2 BPMN 2.0

We chose the language named BPMN 2.0 [174] because in contrast to the first version of this language (1.0), version 2.0 has well-defined execution semantics. A BPMN 2.0 process consists of a set of tasks that are connected by control connectors. BPMN 2.0 has been built with Petri Nets in mind. Thus, it contains additional constructs that are called gateways. Gateways are used to split or join the control flow within a BPMN 2.0 process. Apart from that, a mapping from a subset of tasks of BPMN 2.0 to BPEL has been described in the specification of the language. It is necessary to define a subset of BPMN 2.0 to be mapped to BPEL because even in version 2.0 of BPMN it is possible to define process models that cannot be executed by a workflow engine. It is, for example, possible to create a process model that ends in a deadlock. The mapping from BPMN 2.0 to BPEL implies that BPMN suffers from similar issues with regards to its applicability to heterogeneous service environments and therefore it cannot be used directly in such contexts. The usability of BPMN 2.0 is better than that of BPEL because BPMN comes with a standardized graphical notation with well-defined execution semantics. The Object Management Group (OMG) standardized BPMN 2.0.

2.3.2 Static approaches

Static approaches (also known as syntactic or manual) require the designer to build an abstract process model out of which a specialized framework will produce a service composition. That model includes a set of tasks and their input and output parameters. A task may contains a query clause that is used to search for the real atomic Web service that fulfills the desired task. At runtime, a framework does the selection and binding of the corresponding atomic Web service automatically using a service repository such as the Universal Description, Discovery and Integration (UDDI) [41]. The most commonly used static method is to specify the process model as a directed graph. An example of such an approach is Eflow [28]. As such, Eflow employs a search recipe in order to resolve which search is needed for each node at runtime. Another approach to static composition is proposed by Gronmo et al. [70]. The authors propose using Unified Modeling Language (UML) as the integration platform for Web service composition. The method utilizes a two-way mapping between WSDL service description and a WSDL-independent service model in UML. This mapping allows automating the transformation of WSDL description to UML models and vice versa. The weight in this approach falls into the discovery of appropriate services. Once the service has been discovered, the process of composition is done manually via UML modeling.

A special variant of static composition produces service composition model expressed in workflow language such as WS-BPEL or BPMN out of the abstract model described previously by the designer. WS-BPEL and BPMN are XML-based standards used to describe the specification of Web services, their flow composition and execution. By default WS-BPEL and BPMN use static service discovery, which is implemented by the partner link element. This means that every execution of an external service is tightly coupled at design-time to a particular external service at XML/XSD level. In order to overcome this limitation for the purposes of service composition, dynamic, query language based [125], or even adaptive publish-find-bind approaches [112] are used at runtime in order to permit the underlying framework responsible for the execution of the workflow to query the UDDI and allow for runtime selection and substitution of services.

A further improvement found in the state-of-the art for static service composition is the usage of 'semantic suggestions' for service selection during the composition process; the designer still needs to select the service required from a shortlist of the appropriate services and link them up in the order desired. This variant is often referred to as semi-automatic web service composition. Sirin et al. [151] propose a system that provides service choices that are semantically compatible at each stage. The generated workflow is then executed. Cardoso and Sheth [27] propose a framework that provides assistance to the user by recommending a service meeting the user's needs. This is done by matching the user specified Service Template (ST) with the Service Object (SO).

A further improvement found in the state-of-the art for static service composition is the usage of 'semantic suggestions' for service selection during the composition process; the designer still needs to select the service required from a shortlist of the appropriate services and link them up in the order desired. This variant is often referred to as semi-automatic web service composition. Sirin et al. [151] propose a system that provides service choices that are semantically compatible at each

stage. The generated workflow is then executed. Cardoso and Sheth [27] propose a framework that provides assistance to the user by recommending a service meeting the user's needs. This is done by matching the user specified Service Template (ST) with the Service Object (SO).

2.3.3 Artificial Intelligence Planning

From an Artificial Intelligence (AI) planning point of view, web service composition can be perceived as the construction of a process to attain a specific goal. As such, the problem of service composition is transformed to an AI planning problem that has been extensively investigated by the AI community since the early days of AI. The definition of an AI planning problem is given by Russel et al. [142]; Given a description of the initial state of the world, a description of the desired goal, and a description of the possible actions that can be executed, the purpose of an AI planning algorithm is to construct one or more paths that lead to a certain goal. More formally, this problem can be described as a tuple with five elements: (S, S_0, G, A, G') where S is the set of all possible states of the world, $S_0 \subset S$ and denotes the initial state of the world, G' and denotes the goal state of the world, A is the set of actions the planner can perform and G' is the relation $G' \subset$ S x A x S.

The following subsections describe different computation models that can be used for AI planning of service composition. These computation models are: Finite State Machines (FSM), Situation Calculus, Hierarchical Task Networks and Petri nets.

2.3.3.1 Finite State Machines

A FSM is a model of computation, that models behavior of a finite number of states and transitions between those states and actions. Berardi et al. [15] focus on services whose schema can be represented as a FSM and propose an algorithm that checks for the existence of a composition. Bultan et al. [22] utilize a Mealy Machine; an extended FSM that generates an output based on its current state and additionally an input. According to this approach, services communicate by sending asynchronous messages and each service has a queue. A global watcher keeps track of all messages. The conversation is introduced as a sequence of messages. By studying and understanding conversation properties, this method provides new approaches for designing and analyzing well-formed service composition.

2.3.3.2 Situation Calculus

Situation calculus is a logic formalism introduced by McArthy [106]. It is used to model a dynamic world as a progression of a series of situations, consequences of various actions performed within the world. The calculus is consisted of three elements: the actions that can be performed in the world, fluents that describe the state of the world and the situations. Narayanan et al. [117] provide a limited set of assumptions, under which Web service composition can be realized as a process of reasoning about action formalisms modeled in Situation Calculus. A popular tool when working with Situation Calculus is Golog [98]; a high level logic programming language, built on top of Situation Calculus, for the specification and execution of complex action in dynamic domains. McIllraith et al. [110] propose a method that makes uses of software agents that deal with reasoning about Web services to perform automatic Web service discovery, execution, composition and inter-operation. The proposed method is built on an extended version of Golog that provides knowledge and sensing actions in order to become a suitable formalism for representing and reasoning about the Web service composition problem.

2.3.3.3 Hierarchical Task Networks

Hierarchical task network (HTN) planning [97] is a method of planning in the form of task networks. As such, it differs from other planning methods since it focuses on tasks instead of states. An HTN based planning system operates in a continuous loop. In the duration of this loop, a task is constantly decomposed into a set of sub-tasks until the resulting sub-tasks are decomposed to such an extent, that they have become primitive tasks and can therefore be executed directly. In each step, during this process of decomposition, tests are made in order to check if conditions are violated. The continuous loop terminates if that is the case, if a condition is violated or if the process of decomposition has reached the level of primitive sub-tasks. HTN planning is re-purposed by Sirin et al. [152] for the process of Web service composition in a tool called SHOP2 that uses DAML-S Web service descriptions.

2.3.3.4 Petri nets

A Petri Net (short for Place/transition net) is a connected and bipartite graph. It is consisted of places, transitions and arcs. Arcs run from a place to a transition or vice versa and never between places or between transitions. The places from which an arc runs to a transition are called input places while the places to which arcs run from a transition are called output places of the transition. A Petri Net may fire if it is enabled; when there is at least one token in every place connected to a transition. Hamadi et al. [77] re-purpose Petri nets for Web service composition. In their approach services are modeled as Petri nets by means of mapping. Petri net transitions are mapped to methods and places are mapped to states. Each service, represented as a Petri net has two ports: one input place and one output place. At any given time, a service can either be non-instantiated, ready, running, suspended or completed. After the transformation of each service to a Petri net, composition operators are used in order to combine different services.

2.3.4 Domain specific language based approaches

The approaches presented in this subsection, rely on Domain Specific Language (DSL)s that aim at capturing richer aspects of service composition that go beyond the plain structure of the messages exchanged, thereby associating semantic context to messages. These approaches are Semantic Annotations, the Planning Domain Definition Language (PDDL), Knowledge-based, rule-based, pattern-based, Agent-based and finally Service Level Agreement based.

2.3.4.1 Semantic Annotation

Web service descriptions such as WSDL are limited into describing only the essential parts that are need by SOAP for the purposes of message exchange between the server that holds the endpoint of a service and a client that consumes that service. Semantic annotations aim at compensating for this deficit by permitting the association of meta-data to Web service descriptions by means of ontologies. This association enables coupling a multitude of Semantic Web languages to Web service descriptions. A few examples of ontology languages are DAML+OIL [85] and OWL [109]. When it comes to Web services languages OWL-S [104] (formerly DAML-S [7]) can be used which is an extended version of OWL. On top of OWL's main constructs, OWL-S has a Service Profile, a Process Model and Grounding to describe a Web service. These additional constructs can be used by software agents for the purposes of discovering and using a service. The METEOR-S framework, proposed by Patil et al. [130] is the most cited approach for semi-automatic annotation of Web services.

2.3.4.2 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language is a widely recognized standard used by planners that has influenced DAML-S and as such it can be very easily mapped to and from DAML-S as long as only declarative information is concerned. In 1998 McDermott et al. [107] introduced the first version of this language, which has since then reached to version 3.1 [92]. The obvious advantage here is that as long as the designer uses this language, any PDDL-compliant planner can be used in the process of composition. To deal with Web services specifically, McDermott et al. [108] introduce a new persistent knowledge type that is called a truth literal. This allows the system to interpret that old states disappear and new states are produced.

2.3.4.3 Knowledge-based

Knowledge based composition is a similar process to that used by expert systems such as CLIPS [64]. In such systems, a domain-specific language is used to formulate a common vocabulary that captures the terms that represent an experts understanding of a particular field. In the case of service composition, the field is that which defines the properties of the different services to be composed, for example, if we are looking into knowledge-based composition for the purposes of traveling, then the particular language will contain constructs related to reservations, means of transportation etc. An example of knowledge-based composition is provided by Chen et al. [31]. In this work, the authors utilize domain specific knowledge to guide the process of service composition.

2.3.4.4 Rule based

Rule-based approaches also employ domain-specific languages in order to describe sets of antecedents and consequents to govern the process of service composition. Medjahed et al. [111] propose a composability model that checks which Services can interact with each other. The proposed model has four phases. The specification phase uses Composition Service Specification Language (CSSL) for the description of composite services. The matchmaking phase uses a set of composability rules to describe that different matches among services based on their composability. The selection phase ranks the different matches that have been in the previous phase in order to select the most appropriate one. Finally, the generation phase outputs the description of a composite service. Other commercial rule based approaches such as DROOLS [13] employ RETE algorithms [58] in order to identify very efficiently which rules are affected by the specific event at hand. As such they can be used for the purposes of event-based service composition.

2.3.4.5 Pattern-based

While rule-based approaches are based on logical expressions, pattern-based approaches employ techniques such as software patterns, workflow patterns, collaborative filtering and statistical expression, to formulate characteristics that when identified should result into certain actions. Examples of pattern based composition are reported in [163, 178, 30, 162]. What is interesting to note regarding these approaches is that the process of service composition, through patterns, acquires a more predictive facet and as such it is no longer limited to pre-defined constructs and can therefore utilize probabilistic techniques.

2.3.4.6 Agent-based

Mobile agents, like most agent-based approaches make use of autonomous entities called agents in order to achieve a goal. In some cases the goal may be achieved by an agent work on their own while in other cases when different agents are collaborating towards the same goal. The usage of mobile agents is encountered in Ermolayev et al. [53] work. The authors here use semantic annotations in order to describe a common language (therefore a protocol) that is used for the purposes of communicating between different agents. In this context, an agent has the capability of discovering services by means of a registry or by asking other services in a peer-to-peer manner. The main benefit of the proposed approach is that service composition is no longer centrally governed by a workflow. However, the added benefit comes at the expense of additional message exchange. Cheng et al. [32] further refine on the usage of agents in service composition by proposing an Agent Service Description Language (ASDL) for describing the behavioral characteristics of a Web service, as opposed to its external behavior that is described in WSDL. In this context, behavior refers to the permitted sequence of invocations that are allowed by an agent.

2.3.4.7 Service-Layer agreement based

Service layer agreement based composition or SLA-based composition employs policies expressed in domain specific languages in order to define the required characteristics of a service composition. This notion is used in [136, 37]. The common characteristic of such frameworks is the usage of breadth-first or depth-first algorithms during the process of exploring the search space for the possible alternative implementations of a composite service that lives up to the requirements described by the SLA. As such they utilized dynamic algorithm that may not always yield a solution in PTIME [127].

2.4 Summary

There is an abundance of service composition approaches in the state of the art. This section aimed at identifying their key characteristics for the purposes of better understanding what is currently available. Based on those characteristics, existing approaches are grouped in the following categories:

1. Static approaches
2. Artificial Intelligence Planning
3. Domain-specific language based

Service composition approaches used within the telecommunications industry relate more closely to Static approaches since they model the process of service composition manually. These approaches are Distributed Feature Composition, Initial Filter Criteria, Intelligent Networks and JSR116/JSR289. The key benefit of such approaches is explicit control, since the developer of the composition can very directly control all the aspects of service composition, such as the execution flow and the different services that are being used. Techniques used for Charging and Policy charging and rules functions (PCRF) are more closely related to the domain-specific language approach.

The Domain specific language based category is split into seven subcategories: Semantic Annotations, the Planning Domain Definition Language (PDDL), Knowledge-based, Rule-based, Pattern-based, Agent-based and finally Service Level Agreement based. The approaches within this category examine service compatibility, take into account both functional and non-functional requirements and allow reasoning on which component is best to use in each situation.

Finally yet importantly, the artificial Intelligence planning category is split into four subcategories based on the different computation models that are being used: Finite State Machines, Situation Calculus, Hierarchical Tree Networks and Petri nets. As such the approaches in this category deal with the problem of service composition as a planning problem, where the formal properties of the aforementioned computational models are used in order to define the sequence of atomic services which when combined can achieve a certain goal.

Chapter 3

The approach

Our approach to the problem of service composition is the following:

- Identify the requirements for service composition in a heterogeneous service landscape (section 3.1)
- Assess and discuss to which extent these requirements are met by the state of the art approaches presented in Chapter 2 (section 3.2)
- Based on the previous assessment provide a set of guiding concepts and central objectives for the proposed approach (section 3.3)

3.1 Service composition requirements

The list of requirements has been compiled by aggregating challenges in service composition both from the academic state of the art and also from the telecommunication industry. More specifically, through the process of interviews, the experiences of Ericsson software engineers who have been actively working in system integration activities in the telecommunications domain was recorded. Out of this process three major requirements have immersed; the need for unified application routing (3.1.1), support for concurrency (3.1.2) and last but not least more flexible type checking techniques (3.1.3). The following subsections further elaborate on these requirements.

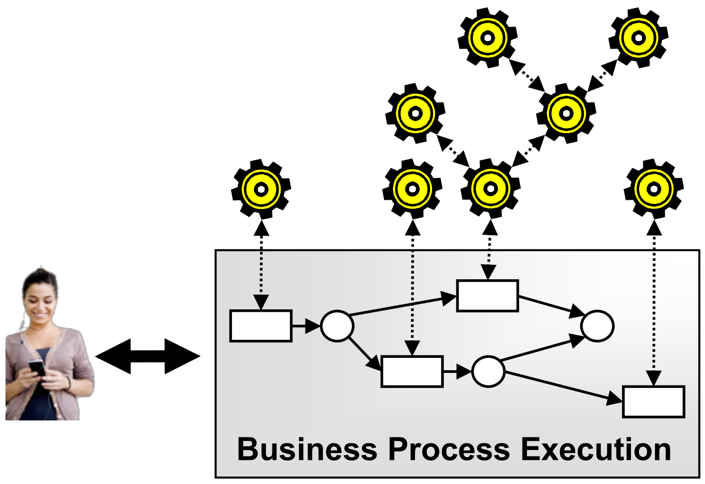

FIGURE 3.1: Service usage in Web service orchestrations

3.1.1 Unified application routing

Application routing as applied in telecommunication networks such as IMS is fundamentally different from web service composition [18]. In order to find the key requirements on a converged composition technology, it is necessary to better understand the similarities and differences between the two approaches.

In web service composition, web services are invoked following a request-response scheme. Requests are explicit and target a specific web service to be executed. A web service may be composite and as such it can be consisted of additional atomic web services. Therefore, web service composition as a whole can be illustrated as a hierarchical tree structure, with the composition process at each level under the control of a single business entity (Figure 3.1). Parallel processing and service requests, which may be received and processed by participants of the composition in any order or even in parallel, are possible as long as explicit execution ordering dependencies are fulfilled. Ultimately, all services are invoked to serve one particular user request.

In contrast, within the telecommunication domain multiple parties or users are in a peer-to-peer relationship. This is reflected in telecom applications being composed as a chain structure (Figure 3.2) (also reffered to as service chain). This chain represents an end-to-end signal flow between telecommunication session participants, with the services allocated as logical nodes on the chain. Each service in the chain has a persistent connection, i.e. a SIP dialog, with its immediate neighbor without

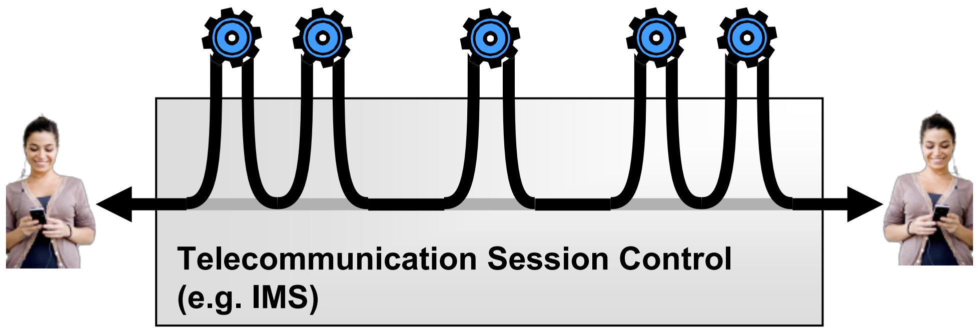

FIGURE 3.2: SIP service chaining

being aware of its neighbor's function. Thus, the service participates actively in the session signaling. In order to manage this service chain, sessions that span from endpoint to endpoint and across multiple nodes and networks are a central concept.

When establishing a telecommunication session between multiple parties, no services are explicitly specified. However, each of these parties may have services defined according to their particular needs. This view is user (endpoint) centric. The services that a user is subscribed to are invoked implicitly during session establishment. The application chain may traverse multiple administrative domains, thus different nodes might compose different segments of the chain.

Furthermore, multiple parties may imply competing requests and multiple services may be invoked on the same triggering condition resulting in overlapping functionality. In this respect, chains represent a very sequential structure and correct ordering of services is essential for service interaction. For a unified service composition approach this means that the session concept, with its characteristic logical allocation and control of services, and a possibility to solve and control service interaction, needs to be supported next to the explicit invocation in a request-response scheme.

Within a telecommunication context, there are real-time requirements originating in the timing requirements of SIP as well as in user requirements. For example, post-dial delay (the duration that extends between the end of dialing and receiving a call progress tone) is typically bounded. In contrast, a best-effort response time is typically required from web services. Predictable behavior regarding non-functional requirements, and therefore their control, is needed within converged composite applications.

In the web domain, web services can be invoked synchronously or asynchronously. In either case, the composition node is maintaining the interaction relationship with its invoked modules for the duration of their respective invocations. Furthermore, they can be invoked in parallel or in an order that best satisfies the requirements of the implemented application. The services are typically not aware of each other, thus they do not interact directly with one another, but only through the central composition node. For a controlled interaction with the services, the composition node maintains state across service invocations, whereas a module it invokes normally does not maintain state after returning. In this way, a module can be seen to exist only for the duration of its interaction with the composition node.

In contrast, telecommunication services are invoked sequentially and operate asynchronously. An application router is invoked by reception of an initial SIP message. It invokes a service by forwarding this initial SIP message to the service. The next service in the sequence will be invoked after the previously invoked service issues an initial request to the application router. Subsequent to their invocation, the services exchange SIP messages in-between themselves to directly interact with each other. A central application router is only needed in the session establishment. Thus, it usually does not need to maintain extensive session state. The services themselves contain state related to the SIP signaling channel they are participating in.

What emerges from this comparison is that the process of service composition for web services and telecommunication services are significantly different. An application router invokes its telecommunication features in a sequence, considering features interaction via messaging over a SIP signaling channel that the application router is not privy to; and composition response time has soft real-time constraints. On the other hand, a composition node for web services interacts over the lifetime of the services with messages exchanged only between a service and the composition node, not between services. Web service composition response time is best effort.

With an understanding of how these two approaches differ, we can now identify what is required from a converged composition mechanism:

1. A mechanism that permits interaction with a persistent telecom service after having returned from invoking it. This motivates the introduction of the ”composer” logical functionallity. As such this would enable the composer to

be informed of state updates that may occur in the telecommunication domain in response to receiving SIP messages along the separate signaling channel. Moreover, it would enable the composer to influence both telecom and web modules after their invocation.

2. A lightweight module invocation mechanism that supports real-time limits.
3. Support for managing service interaction that naturally arises amongst composed services.

On top of these requirements, telecommunication network operators need extensive control over the services they allow to be provided through their network. Controlled service behavior and service quality are important in this respect, in order to keep high standards regarding reliability and availability of a single service offer or the entire network. Furthermore, inter-operability between networks and nodes need to be ensured by following appropriate standards.

3.1.2 Concurrency

Concurrent computing is as a well-known form of computing in the state of the art, where programs and functions thereof are expressed by means of processes or threads and may be executed in parallel. In most cases parallel execution is either achieved via time-slicing techniques or by utilizing one or more processors that are assigned with different tasks (or computations) to be executed at the same time. As a consequence, communication between different processes or threads becomes an important aspect.

There are two main paradigms with regards to explicit communication, shared memory and message passing communication.

- Shared memory communication is exemplified in languages such Java and C# and allows for communication between processes via altering contents of a common and mutually shared memory space. To avoid unwanted changes techniques such as semaphores and mutexes are used to coordinate between threads.

- Message passing communication has become popular through languages such as Erlang and Scala. In such languages messages are exchanged asynchronously – this defeats the need for locking shared resources and is considered more robust than shared memory communication.

Several concurrent programming languages exist, D [4], Comega [17], Erlang [10], Orc [90], Plaid [3], Scala [121] are just a few examples. For a complete list of concurrent programming languages the reader can refer to [14].

Plaid is particularly interesting from our perspective since it is a concurrent by default systems language that leverages multicore systems out of the core. Even though concurrent programming and concurrent models of communication are well known in the state of the art, most frameworks for workflow languages do not utilize said techniques and therefore offer decreased application throughput, low responsiveness with regards to I/O and cumbersome constructs for crafting concurrent programs.

Sutter [149] elaborates on one of the most fundamental technological shifts in the last few decades. Since it is no longer possible to improve single Central Processing Unit (CPU) performance, hardware vendors have started to integrate multiple cores in a single chip. Effectively this forces a developer to build concurrent applications in order to achieve performance improvements in new hardware. However, current programming paradigms build in sequentially and as a result, concurrency support in those languages forces developers into low-level reasoning about the execution order. At the same time, writing concurrent applications is notoriously complicated and error-prone, because concurrent tasks need to be coordinated to avoid problems like race conditions or deadlocks.

Consequntly, inherent support for concurrency becomes a requirement itself in order to offload a developer from the need for catering herself for schedulers that deal with concurrent tasks and message passing styles for communicating input/output parameters between such tasks.

3.1.3 Type checking

A type is a set of values that have common operations (i.e. keyword ”int” in langauges such as Java represents a domain of integer numbers). Type errors occur when an operation is made to a value of the wrong type. From a type checking point of view programming languages can be split in two main categories, dynamically or statically typed. Cardeli et al. [26] provide a more detailed categorization of the different variations found in the area of type systems but for the purposes of describing this requirement we will only focus on these main two categories since they represent the two opposite ends in the spectrum of type systems. In dynamically typed languages the operation of type checking is performed during program execution (runtime). In statically typed languages the same operation is performed at compile time. When, experimenting with a new piece of software, developers appreciate dynamic type checking because it does not get in their way; the developer does not need to adapt the program to the type checker. In addition, dynamic type checking makes it easy to deal with situations where the type of a value depends on runtime information. At production time, static type checking is mostly appreciated because it allows the developer to catch bugs earlier, thus reducing the cost of fixing bugs later in the development cycle. In addition, static type checking enables faster runtime execution due to fewer checks.

Consequently, the requriement here is the ability to choose and even mix between dynamic and static type checking. Such a feature would permit a designer to benefit from dynamic type checking at design-time while suffering from the inherent perfromance impact and thereafter move the same piece of functionallity to static type checking, therefore improving on performance.

3.2 Assessment and discussion

This section discusses the impact the requirements identified in the previous sections have on service composition. The impact is assessed per category by taking as input the categories to service composition approaches identified in Chapter 2. These categories were Static, Artificial Intelligence Planning and Domain-specific language based. The following subsections provide detailed assessments for each category.

3.2.1 Static approaches

The key advantage of static (also known as manual) composition is that it grants the designer, explicit control of the service composition process both at design-time and at runtime. This is achieved at the expense of flexibility by using simplistic mechanisms for service selection (query languages) and for describing the execution flow. Along this line, manual service composition techniques in conjunction with service usage patterns seem to be promising approaches for a commercial deployment of service composition activities in a telecommunications operator. Analysis of service usage patterns can identify sets of services, which are good candidates for the creation of new composite services. Consequently, a manual process can initiate the conversion of these services into UML models and the creation of a composite model that in the end is transformed into a new executable composite service.

In most approaches, the resulting service composition model is described in a domain specific language that is later on converted to WS-BPEL or BPMN to be executed by the corresponding framework. Examples of such frameworks are ActiveBPEL [1], Microsoft Windows Workflow Foundation [148], Oracle BPEL Process Manager [122], Oracle SOA Suite [89], IBM Websphere [20], RedHat jBPM [40] and Apache ODE [155] and others.

However, these approaches suffer from the following drawbacks that relate more closely to the runtime aspect of service composition:

1. The first issue identified with such frameworks is that the workflow can only interact with services that are either directly or indirectly exposed as Web Services following the WSDL standard. This limitation is rather crucial to the telecommunications domain, where services are made available via several other protocols such as SIP and the Signaling System No 7 (SS7) [45].

2. The second issue is that these frameworks employ a static type system. This means that at design-time, the development environment would mark the workflow as invalid if a type error occurs and at runtime the framework would abort execution completely, in case of a type mismatch. This is an important issue that hinders productivity since it can very well be the case that at design-time the designer is not aware of the exact types the workflow is going

to use. Moreover, if we consider dynamic service selection, it can very well be the case that the input towards an external service, the one that the designer had in mind when developing a composition, is not the one expected by the service that will ultimately be selected at runtime. Hence, the expectation that the execution framework should be able to make corresponding adaptations so that such issues could be avoided in a best-effort manner. This limitation clearly impacts the requirement for using a flexible type system as detailed in 3.1.3.

3. The third issue relates to the inherent difference between orchestration in the telecommunication domains and in the web domain. WS-BPEL frameworks as such are geared towards the Web and as such they are not suitable for telecommunications as detailed in the requirement for Unified application routing in 3.1.1.

4. The fourth issue is found from a parallelism and concurrency perspective. In a workflow one needs to define execution of parallel tasks explicitly. This increases the amount of effort for designer because he needs to use additional constructs, implement logic in order to make the workflow capable of utilizing multiple cores, which is a standard in most computer systems these days. Moreover, Figl et al. conducted a study in [56] which shows that workflows designed in languages such as WS-BPEL, BPMN 2.0, UML [46] and Yet Another Workflow Language (YAWL) [164] are hard from a cognitive point of view due to the usage of routing symbols such as split, join, decision and merge. This limitation severely impacts the requirement for concurrency detailed in 3.1.2.

5. The fifth issue is a common limitation found both in WS-BPEL frameworks and the DFC approach. The assumption here is that services or workflows do not share state and they do not know the existence or depend on other services or workflows. This assumption, from the one hand side allows for efficient mechanisms for the creation of simple workflows. On the other hand side the lack of a common state makes the implementation of more complex workflows cumbersome since state information needs to be transported from one service to another during the execution of the workflow. This unnecessarily increases the amount and complexity of signaling between services/workflows.

3.2.2 Artificial Intelligence planning

The advantage of explicit control over the service composition process is sacrificed in AI planning approaches to make way for the added benefit of automation. More specifically AI approaches aim at automating the service composition process at design-time. Pistore et al. [131] demonstrate an example where the composition of a bookstore application is done automatically in a matter of minutes where the same application requires 20 hours of manual work for an experienced developer. The main benefit of AI planning approaches is that they are suitable for modeling the inter-workings of a complex service. Although the ideas expressed within the various AI methodologies are very promising, they are difficult to apply in real world environments for three reasons:

1. The first reason is the usage of a closed world assumption [23]. This assumption works nicely when the designer of the service composition process has direct control of the implementation for the services that are going to be composed. When this is the case, the designer knows both the effects and the side effects of such services and the AI planning algorithm can take these characteristics into account or alternatively, the implementation can adapt to the characteristics of the formal model that governs the AI planning algorithm. However, if that is not the case, then the side effects of external services are not known since they cannot be described in a WSDL document. In such cases the AI planning algorithm can still be applied but without any guarantee that the resulting service composition will be useable in practice.

2. Srivastava et al. [154] discuss further on the limitations of AI planning approaches. Their main argument is that AI planning approaches are not applicable to Web service composition, due to the assumption that all objects, related the problem of composition are available in the initial state for the planner to use, which is not the case with Web services, since new objects can be generated at runtime, thus causing the AI planner to restart and/or increase the search space.

3. Another issue encountered in AI planning approaches is observed in the frameworks [133] used in such approaches. More specifically, such frameworks are not efficient in the sense that if a particular service implementation is not

available in the service repository, planning of service composition will stop and fail, without producing or storing for later use a partial result. Additionally, such frameworks are deprived of a composition repository that would make it possible to reuse previously constructed compositions.

As such AI planning is a very useful feature to have at design-time in order to generate recommendations for the designer, provided that recommendations are simple and human readable. However, from a computation complexity point of view it is not efficient to apply such techniques at runtime in the process of producing on the fly a composition of services once an external request, a trigger to a composition has been received.

3.2.3 Domain-specific language

Domain specific language approaches are tailored to excel in the particular application domains they are designed to support. The term excel here is used broadly to accentuate the fact that such languages are consisted of only the required set of constructs needed to describe the composition at hand and nothing more and thus expressions written in such languages end up being less verbose than general purpose languages that could have been used to describe the same composition. More specifically, semantic annotations describe service compatibility, take into account both functional and non-functional requirements and allow reasoning on which component is best to use in each situation. They can easily find alternates to existing services and replace them if needed.

However, they are designed to allow description of service in a machine-readable way, thus making it difficult to be used by humans. Knowledge-base approaches rely on capturing knowledge from experts regarding the resolution of a particular problem and as such proceed with the process of composition in a Q&A fashion that aims at limiting the search space and as a result produce a composition. The key difficulty here is the amount of time it takes for an expert to capture intuition and technical knowledge in a machine-understandable format. Rule-based and pattern based approaches are similar in the sense that they both follow an ”if-then-that” pattern (also known as ”when-then”) where the first part of the rule (or pattern) is known as the antecedent and the second part as the consequent [69]. Despite the high performance that rule-based and pattern-based approaches enjoy [99] the main criticism comes with regards to the maintainability of several rules accumulating in a system over time.

In conclusion, in domain specific language approaches interoperability between different compositions is sacrificed for the added benefit of laconic expressions. On the hand side this limitation is justified since the development of domain specific language for multiple application domains is particularly challenging.

3.2.4 Assessment summary

Table 3.1 summarizes the assessment of the three approaches to service composition as it has been detailed in the previous subsections. The assessment is split between design-time and runtime aspects. For design time the different aspects used in our comparison are design to runtime relationship, formal semantics, runtime objects and service availability. Design to runtime relationship describes the relationship between the model that the developer of the service composition creates while designing the composition and the model that is produced by the virtual machine to be executed at runtime. Formal semantics describes whether or not a formal model exists detailing the execution semantics of the language used to describe the model for service composition at design-time. Runtime objects describes whether or not runtime objects are tolerated by the mechanism that produces the service composition model. Service availability relates to the amount of information that is required in the service repository in order to produce a service composition and whether or not that limits the service composition process.

Moving on to runtime aspects we used service invocation, type system, unified routing, concurrency and shared state. Service invocation is about any limitations that may exist when the service composition model communicates with external services. Type system is about the usage of a static, dynamic or optional type system as explained in 3.1.3. Unified routing is whether or not unified routing as explained in 3.1.1. Concurrency is about how concurrent processes are defined within the service composition model as described in 3.1.2. Finally yet importantly Shared sate is about support or lack of a shared state when exchanging information among different composition models at runtime.

		Static approaches	AI planning	Domain Specific Languages
Design-time	Design to runtime relationship	Direct (uses common language constructs to define composition)	Implicit (planning problem is transformed to a composition)	Indirect (domain specific constructs are related to general purpose instructions)
	Formal Semantics	Most cases	All cases	Most cases
	Runtime objects	New objects can be created at runtime	All objects are available in the initial state – new objects are not tolerated at runtime	New objects can be created at runtime
	Service availability	Lack of suitable services in the service repository would break the process at runtime	Lack of suitable services in the service repository breaks the planning process at design-time	Lack of suitable services in the service repository would break the process at runtime
Runtime	Service Invocation	Limited to a particular technology		
	Type System	Static type system		
	Unified routing	Not supported		
	Concurrency	Explicit definition		
	Shared State	Not supported		

TABLE 3.1: Assessment of approaches to service composition

The key difference between the approaches we have assessed so far is found in the design to runtime relationship. All three approaches use their own unique ways for describing the service composition model at design-time. Static approaches use constructs inspired from general purpose programming languages (i.e. for/while loops, if statements etc.) to describe the composition, AI planning techniques abstract away from how the composition is formed and focus more towards what the composition should do and domain specific language approaches use yet another abstraction, inspired by the corresponding domain, to describe the composition model (i.e. a service composition for a medical service is likely to contain constructs relevant to the medical field).

Another discrepancy we observe is that the usage of formal semantics is not that widespread when examining approaches within the static and domain specific language areas. In most cases, lack of formal semantics is considered as bad practice since it makes hard to evaluate such approaches due to the lack of a formal model.

Runtime objects and service availability seem to pose limitations mostly in AI planning approaches since in most cases planning algorithms require as much as possible a-priori knowledge of the planning problem in order to limit their search space.

When visiting the runtime aspect of all three approaches we encounter the same type of limitations mostly because all three approaches tend to translate their corresponding service model to one that is more similar to the ones produced by static approaches since the properties of such models can be more easily converted to runtime executable service composition. As such we find limitations in service invocation, type system, unified routing, definition of concurrency and lack of a shared state.

3.3 Guiding concepts and central objectives

This section introduces the central objectives and properties of the proposed service composition approach and outlines its key concepts. Our approach consists of a framework that is responsible for executing a service composition at runtime and a Service Composition LanguagE, abbreviated as SCaLE that allows a designer to describe a service composition at design-time. Our aim is to design the proposed approach in a modular fashion in order to avoid the obvious caveats of monolithic design and enable bits and pieces to be reused as much as possible. The key concepts of the proposed approach are the following:

1. Technology agnostic composition
2. Service oriented approach
3. Dynamic service selection
4. Asynchronous by default — abstract control flow

5. Events, messages and Shared State
6. Optional type system

3.3.1 Technology agnostic composition

To fulfill the requirements for Unified application routing as described in 3.1.1, the proposed approach to service composition of heterogeneous services should be able to handle both incoming and outgoing requests, from and to the proposed composition framework, in a unified fashion. This key concept aims at overcoming the limitation found in WS-BPEL which confines the designer to services exposed via WSDL. When using the proposed approach, the designer should be able to describe compositions where the constituent component services can be based on a broad range of technologies and not only Web services. Figure 3.3 offers an illustration of the proposed heterogeneity. Examples of such technologies are Asynchronous JavaScript and XML (AJAX) [62], Enterprise Service Bus (ESB)s [29], IN/Customized Applications for Mobile Networks Enhanced Logic (CAMEL) and IMS/SIP technologies. In addition, it should be possible to extend the proposed approach to support future service technologies.

Although these technologies might significantly differ the proposed approach should be able to compose them together, thus integrate them. This concept is both a concern of the underlying framework and also of the language for describing compositions.

SCaLE should provide a number of constructs, which allow interacting with a variety of differently implemented services in a uniform way. For example external actions in the constituent service usage can be exposed via events, external data via shared state variables and services can be abstracted using high-level service description. The interaction with external services of any type or technology is broken down to the usage of a few core elements and concepts, which are the same for any type of service.

Especially the abstraction of services plays a central role in hiding irrelevant technological details. Services should be described using general properties. The idea is

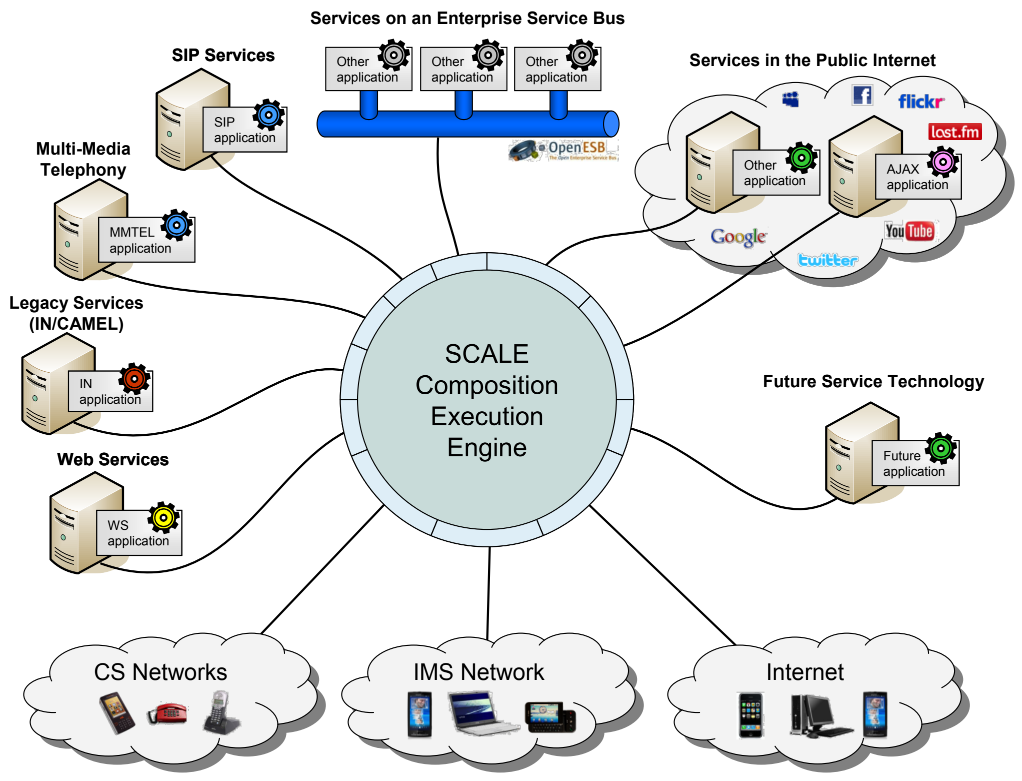

FIGURE 3.3: Composer in a heterogeneous service landscape

that it does not matter how a service is implemented as long as it provides a certain needed functionality.

3.3.2 Service oriented approach

The proposed approach should be designed following the key concepts of a SOA. This includes the notion of a service being a self-contained unit, which provides its functionality through dedicated interfaces. In addition, it includes the concept of a service repository and a publish-find-bind scheme in the interaction between users. Services publish their function to service repositories. Users in need of a certain function find services through the service repository and finally bind the service in order to integrate it into an overall application. Late binding and loose coupling are key concepts in this kind of service environment and should be key properties of the proposed approach.

3.3.3 Dynamic service selection

The designer should be able to use constraints in order to express the required abstract properties of a constituent service, instead of using a hard-coded pre-determined service. It is the responsibility of the underlying framework of the proposed approach that should at runtime, dynamically select a service that fulfills these requirements. Even though this requirement already exists in several well-known approaches to static service composition such as EFlow, it is absent from more mainstream workflows languages such as WS-BPEL.

Dynamic service selection allows for loose coupling and it goes one step ahead of late binding. For simple late binding it is pre-determined which service to use although the actual service is only needed at run time. Dynamic selection does not only perform the binding at runtime, but also the decision which services are used at all.

In addition, the high-level service description plays a central role here. The abstract properties used to describe the available services are also used in order to phrase conditions and requirements. For example, the geographical location of a user might be needed. In order to get this information, the designer would use a constraint that requires a service described with the property ”user location”. The underlying framework should evaluate this constraint at runtime in order to find a suitable service. In general this can be a different service every time the composition is executed, but this does not matter as long as the service fulfills the constraint.

3.3.4 Asynchronous by default — abstract control flow

To fulfill the requirement for concurrency described in 3.1.2, all elements of a composition that is specified in SCaLE should be executed in any order or in parallel unless otherwise it is explicitly stated or implicitly implied by dependencies. This leaves a lot of freedom to the framework that is executing the composition with regards to automatic optimizations gained from a dynamically decided order of execution.

In order to guide the framework the designer can provide information that needs to be taken into account by means of constraints. This can be for example data dependencies implied by one constituent service expecting output of another service as input. SCaLE should provide constructs to model these dependencies. Thus, the control flow in the proposed approach s constraint driven. This implies that SCaLE should be deprived of control-flow elements such as split, join and merge. According to Figl et al [56] languages that are deprived of such elements have a lower graphic parsimony and as such are easier to comprehend.

3.3.5 Events, messages and Shared State

Another central concept of the framework should be events and in particular asynchronous events. Events are the primary trigger for composition and execution. External communication with services is exposed by means of events. Events can contain data, thus they can be used as messages being exchanged between various parts of a composition and between the composition and external services and users. Events in SCaLE accommodate the often-asynchronous nature of interaction between services that can be persistent and independent processes rather than stateless and simple. Especially telecommunication services are often self-contained and stateful entities within a communication session. This requirement aims at overcoming a limitation found in WS-BPEL and DFC, which is that of having a notion of a session and a notion of shared state at runtime.

3.3.6 Optional type system

To fulfill the requirement for type checking described in 3.1.3, the proposed framework should allow for an optional type system permitting both typed and un-typed variables. Un-typed variables can be used while designing and experimenting with the composition in a trial and error fashion; while typed can be used once the model becomes mature, making it suitable to go into production. The designer should be able to interact with the optional type system by using annotations in front of the variables that either defines or not the type of the variable.

3.4 Summary

By taking into consideration the set of challenges identified for service composition and the limitations found in the state of the art, this section motivates a list of guiding concepts that should govern our proposed approach. We identify three challenges that relate closely to the runtime aspect of service composition.

These challenges are Unified application routing, Concurrency and Type-checking. The requirement for Unified application routing is motivated by the intrinsic difference between web service composition and composition in the telecommunication where one is characterized by request-response kind of interactions while the other by service chain patterns. The requirement for Concurrency is motivated by the prevalence of multicore systems and by the complexity of developing customized schedulers that can utilize them. Finally yet importantly the requirement for Type-checking proposes the ability to traverse between dynamic and static type checking techniques in accordance to the maturity of the under-development application.

Having identified these requirements we move on to assess how these challenges are addressed by the state of the art, and overall limitations in the categories of static, artificial intelligence planning and domain specific language based approaches to service composition. Even though static approaches for service composition offer explicit control they have a more limited focus and fail to address aforementioned requirements. Artificial intelligence planning approaches are beneficiary at design-time but at the same time from a computational complexity point of view are not applicable to runtime service composition. Domain specific language approaches offer laconic expressions for formulating service compositions at the cost of maintainability and interoperability. An overview of the assement if shown in Table 3.1.

Consequently, the proposed framework uses a technology-agnostic core composition function, which is comprised of high-level abstraction in the description of heterogeneous services, dynamic service selection and step-by-step instantiation of abstract composite application models. In order to better support telecommunication services, these concepts were combined with support for sessions, integration with the IMS and support for legacy telecommunication technologies such as Intelligent Networks (IN).

Chapter 4

System design

In this chapter, we present the design of the proposed framework and language for service composition implemented, as proof-of-concept of our approach. We outline the objectives of our design, present the relationship of the constituent parts of the framework, and describe each part in detail. In the case of the framework, we describe its constituent components, while in the case of the language we define its constructs, along with a few examples for the purposes of further clarifying how these constructs interact with each other.

4.1 Objectives

The objective of this design and implementation exercise is to try the ideas outlined in the previous chapter on a real system. Our main goals are:

- Provide an experimentation platform, and experiment with service composition
- Identify possible inconsistencies or missing elements needed in our approach
- Demonstrate the viability of our approach

4.2 System architecture

In order to realize the guiding key concepts outlined in the previous section we designed and implemented a service composition framework and a service composition language.

The term framework [135] is used instead of library since our proposal contains the following three properties:

- **Inversion of control** — the flow control is dictated by the framework and not by the caller
- **Default behavior**
- **Non-modifiable code and Extensibility** — parts of the proposed framework are frozen meaning that they cannot be changed and some other parts are hot meaning that they can be extended

Section 4.3 provides a detailed description of the proposed composition framework by describing its constituent components. Section 4.4 provides a detailed description of the composition language. Finally, section 4.5 concludes this chapter with a set of examples that will help the reader to better understand what can be expressed with the proposed language and how.

4.3 Service composition framework

The proposed framework defines a system for composition of converged applications using constituent services from web, enterprise and telecommunications domains, both circuit switched (CAMEL Application Part (CAP)/Intelligent Network Application Protocol (INAP)) and IP (SIP) based, able to control feature interactions across technology borders. An overview of the system is portrayed in (Figure 4.1). The system is based on SOA principles [128], thus all services are considered to be autonomous and loosely coupled units. Composite applications are created as "skeletons", designed as a model of the core business-logic of the application in terms

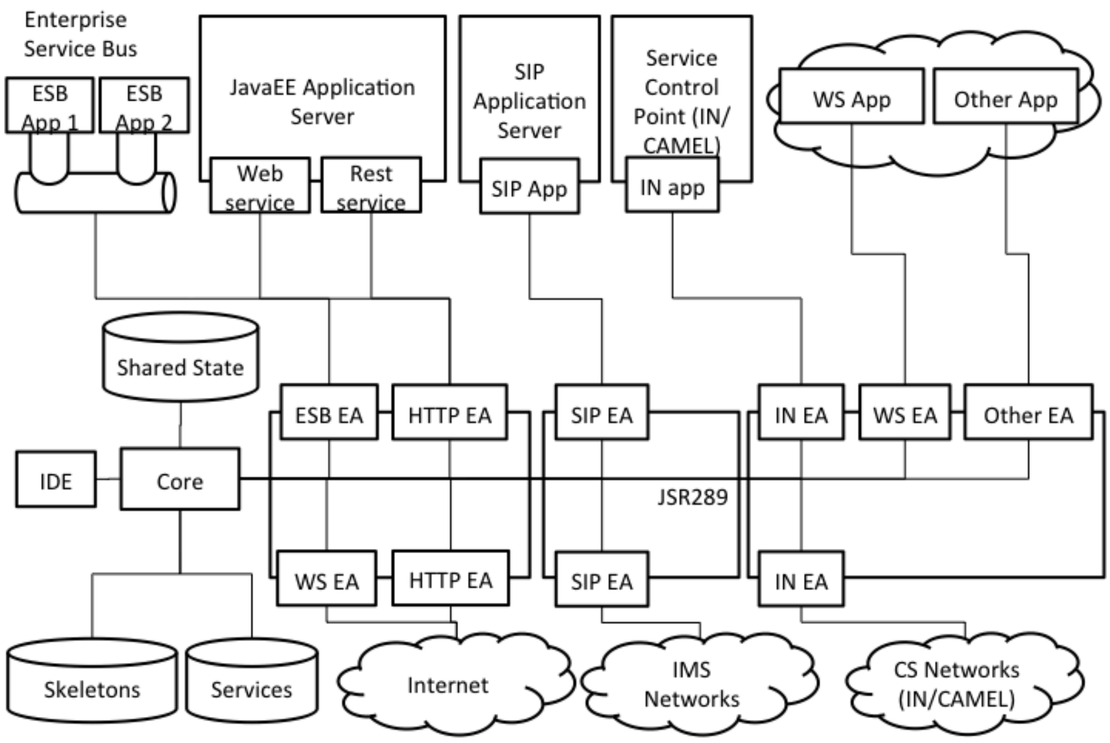

FIGURE 4.1: Service composition framework as a central mediator within a multi-technology environment

of participating constituent services. Protocol-level details related to the interaction with modules are left to the Execution Agent (EA)s, which are responsible for enforcing composition decisions in the corresponding platform in a technology and protocol-specific way. A shared state is used as means of mediating information between the application skeleton and the EAs, thus coordinating the service execution.

The proposed framework is comprised of the following eleven components:

- Composition core
- Application skeleton repository
- Services repository
- Shared state
- Execution Agents

    HTTP

    SIP

    WS (WebSockets [83])

    IN

    Java Business Integration (JBI)

- IDE

The following subsections describe each component in further detail.

4.3.1 Composition core

The composition core is in a way similar to a rule engine, in the sense that it receives external events and decides how they should be handled. This resembles the ”when” part of a rule in a rule engine. Unlike a rule engine, the ”then” part in the composition core is more expressive than that found in rule engines. Section 4.4 details the complete set of constructs that can be used by a designer in order to define what should happen when an event is intercepted. From this angle the composition core behaves very much like an interpreter of a programming language since it translates our source code into an efficient intermediate representation and immediately executes this. In our case the source code is expressed as skeleton and it is stored in the composition repository. When an interpreter has to make a function call, usually that call is defined explicitly and it targets one of the available functions that exist in the current scope. In our case, function calls are actually invocations towards external services. In order to define the invocation of such services we employ a simple query language that queries a list of available service descriptions that are located in the services repository. This part of the process deals with deciding which service to invoke. The remaining part of this process is to actually invoke the external service. That part is delegated to the corresponding execution agent that is responsible for that class of services.

The order in which different steps defined in our source code are executed is defined by a data dependency graph [82] that is computed on-demand (or even pre-computed in some cases). The data dependency graph is formally defined as G=(I,R) where I is a set of instructions and R is a transitive relation R=I x I with (a,b) if the instruction $a \in I$ must be evaluated before $b \in I$.

Events are propagated to the composition core through execution agents. Once an event is detected, a new session is generated. That session is stored in the shared state, which holds all runtime information about this particular instantiation including intermediate variables that may be generated at runtime that hold the results/responses received by contacting external services.

4.3.2 Application skeleton repository

The application skeleton repository stores a collection of application skeletons that are available in our framework. As mentioned in the previous paragraph, an application skeleton (or skeleton for short) is similar to the source code of a program and as such it describes the design-time specification of a composition. As such it is comprised of a set of constructs that describe queries to external services, data dependencies and intermediate variables that may be used at runtime. A more detailed description of the constructs of a skeleton is given in 4.4. An application skeleton is analogous to a graph, as it is already implied in subsection 4.3.1 and as such the schema used in the storage facility for application skeletons consists of a sets of nodes and edges.

4.3.3 Services repository

An essential component of the proposed framework is the use of formal service descriptions for all constituent services. Service descriptions are important for service discovery, selection and invocation. They contain information about the service API and service binding information. In this respect, the information available in service descriptions resembles that which is found in intermediate language descriptions such as WSDL or Web Application Description Language (WADL) [75]. Their key difference is that they are not limited to describing Web or Restful services; services from a variety of technological backgrounds can be described (i.e. REST, JBI, WebSockets [83]). Service binding information is accompanied by a collection of abstract properties that reflect service capabilities and functionality. In principle, the proposed framework does not require specific mandatory properties within the abstract service description, nor does it impose a closed-world assumption. In practice, a property called 'type' is used to describe the function of a service. For example, in order to describe a service that provides the position of a user, the respective abstract description of the service could be 'type:positioning'.

Within the proposed framework, a composite service application is designed by describing the required components in an abstract way. If the composite service developer requires specific functionality as a constituent service within a composite application, she expresses the essential properties of the needed functionality rather than pointing explicitly to an individual service. Thus the developer specifies requirements on the component functionality rather than selecting the components explicitly. The selection of services that provides the required functionality and will therefore be invoked is left to the composition core to determine at runtime. In this way, loose coupling between a composite application and its components is achieved. The requirements on a constituent service are expressed in the form of a query, which is based upon the properties used for describing the services.

As an example, assume that the composite service developer would like to use a service that is provided by Ericsson and offers information on the position of a user; in this case the query 'type=positioning & provider=Ericsson' can be used. Such a query creates an abstraction between the business logic and the underlying technology that is used to implement or provide this external service. The need to distinguish whether the positioning service from this example is provided as a Web service or via SIP is no longer the concern of the composite service developer.

4.3.4 Shared State

Shared state plays a central role in mediating between service technologies. Constituent services are autonomous and can be used together regardless of their technology. They are unaware of each other's technological details and interaction requirements. The shared state is a universal way to achieve interaction, as it is accessible by any constituent service. The application skeleton can implement mediation by means of additional services that, for example, translate between data formats. Thus, shared state and technology agnostic composition core provide a powerful framework for implementing mediation logic. This framework, accompanied by a toolbox of services for routine tasks like data translation, permits the development of extensive inter-work scenarios with minimum effort.

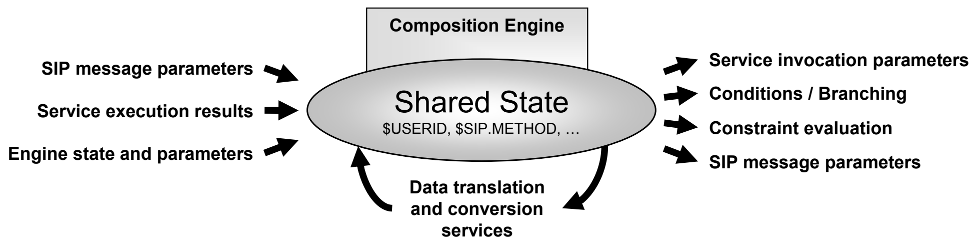
Composition Engine
SIP message parameters
Service execution results
Engine state and parameters
Shared State
$USERID, $SIP.METHOD, ...
Data translation and conversion services
Service invocation parameters
Conditions / Branching
Constraint evaluation
SIP message parameters

FIGURE 4.2: The shared state

Data is important for the inter-work between constituent services and between composition as a whole and external users. Application skeletons utilize variables to store all data exchanged between external users and services. This results in all incoming data being stored into local variables and in all outgoing data being retrieved from local variables. As an example the values of parameters found in the headers and payload of protocol messages are exposed to the composition session as data in local variables and can later on be used as input to constituent services being invoked by the composition core as scripted by the application skeleton.

The entire set of data of a composition session is called ”shared state”. This reflects the usage of data to be shared between constituent services for coordination and as data pool for all inputs and outputs. The concept is shown in (Figure 4.2). In addition, data is shared between services and protocols that belong to different service technologies. For example the user addresses as received via SIP can be used as input parameter to an action that invokes a web service.

4.3.4.1 Locality

Locality of data is a key design characteristic of the Shared state component. Our aim here is to minimize the number of variables where concurrent access may occur without restricting the parallel execution of constituent services using synchronized-blocking calls.

As detailed in 4.4.2.1, an action has local data, which are copied in a message passing style from its parent. Effects of the action are the only way to return values back to the parent. This minimizes the number of possible conflicts and side effects from parallel execution to those variables, which are specified as action effects. Essentially this means that even though the proposed shared state component is not race-condition free; it provides the tools to the designer to formulate the application skeleton in a way that race conditions can be avoided. This notion is inspired by the programming style that is best practice when writing distributed software in languages such as Erlang [10], Scala [121] or even Java [100] where constructs such as a shared state are discouraged and the developer is encouraged to make use of asynchronous and stateless calls.

As a summary, data race conditions within the shared state can be controlled by means of the following aspects:

- Local encapsulation of data copies within nested actions. This separates data from parallel-executed actions.
- Only effects allow the definition of selective writing in parent data. This limits the possibility of race conditions to a small set of variables that are in the focus of the developer.
- Data dependencies allow the explicit selection of the source of input data for an action. This is a tool to control the remaining race conditions of the variables exposed by effects.

4.3.5 Execution Agent

In our approach, the execution of a constituent service is a shared task between the composition core and the EA. As mentioned in 4.3.1, the composition core delegates the task of executing the service to technology specific EAs. The composition core takes composition decisions and the EA enforces them. While the composition core and the process of service selection are independent of the technological details and implementation of a constituent service, the EA executes the service within its specific runtime-environment. An EA is chosen according to the binding information that is a part of the service description. Furthermore, incoming service requests may reach the composition core through one of the EAs.

A variety of EAs has been developed. Each of them provides support for a specific service technology. For example, there is an EA for the HTTP protocol (HTTP EA), which is capable of handling HTTP messages that are addressing either Web Services or Restful Services. Moreover, integration with telecommunication networks and protocols is provided by specialized EAs. The SIP Execution Agent (SIP EA) supports SIP chaining and operates according to JSR289. The IN Execution Agent (IN EA) allows the interaction of the composition with the intelligent network architecture of circuit switched networks by supporting CAP/INAP protocols and allows the composition core to act as a Service Control Function or Service Switching Function [119]. Furthermore, a dedicated EA can be integrated into a JBI-based ESB [36] acting as a service engine.

The composition core exposes to all these EAs one API that is independent of a specific service technology. A complementary API is exposed by EAs and used by the composition core. A new service technology can be integrated by providing a new EA that complies to these APIs. The composition core does not need to be modified. Additionally, this API supports very different ways of inter-working between the composition core and the EA.

The request-response style of service usage, typical to Web services or Restful services, requires a considerably different operation than the end-to-end sessions with service chaining style as known in IMS. In general, the service execution is synchronous. Thus, the composition core waits with further execution of the skeleton until it gets a state update from the EA. For a Web service, this is usually the return value provided as a result of the invocation of that service; for SIP/IMS, this is usually the request for selecting the next service as described in the next paragraph.

The composition core is logically integrated into IMS as a SIP application server through an EA that supports the ISC interface. In the inter-working with IMS, iFC routing triggers the SIP EA resulting in a request to the composition core. The SIP EA acts as an Application Router according to JSR289. It uses the composition core to get instructions about which services need to be invoked within the SIP session and in which order. This processing of SIP services in the application skeleton may be mixed with the invocation of other non-SIP based services, for example with Web services. Their execution is triggered and their results are processed by the composition core according to the skeleton without the SIP EA or the underlying IMS infrastructure, being aware of these activities.

The assembly of the chain of SIP services is a typical activity in IMS at session establishment. The composition core cannot only control this process, but it can also intercept and react on subsequent SIP messages and apply changes to the chain of services. The processing of subsequent SIP messages (non-initial requests or responses) is an enhancement with regards to the capabilities of an Application Router according to JSR289. For example, a session leg can be released from a certain point onwards and the establishment of an end-to-end session can start over at this point. This is necessary if the originally addressed user could not be reached and the session is established with an alternative end point.

The concept of outsourcing protocol-related functionality to external EAs introduces heterogeneity to the presented approach and permits using it for the SCIM [68] as defined by 3GPP.

4.3.6 Integrated Development Environment

The Integrated Development Environment (IDE) (also referred to as service creation environment) is the tool to be used by the designer in order to design the application skeleton. As such it is capable of providing a palette of different constructs that can be used for the purposes of this design. Moreover, the IDE should allow for ”post-mortem” debugging of an instantiation of an application skeleton. This feature is deprived of real-time debugging aspects, but it is capable of portraying a picture of the steps and how the were executed after the composition core has finished processing the application skeleton.

4.4 Service Composition LanguagE

The design of a language should be as much as possible independent from the underlying framework that is responsible for its interpretation. However, some concepts demand a common understanding of a few high level architectural concepts needed within a suitable implementation of an interpreter. The objective to stay technology agnostic at the language level is addressed by using abstract services in a SOA sense. These services are defined by abstract properties describing the function they provide. These services are also selected based on these abstract properties. This allows the designer of the composition to operate on a high abstraction level. The designer can focus on the overall function instead of each and every technological detail.

However, somewhere the abstract service needs to be evaluated and interpreted based on the concrete technology of its implementation. This is achieved by an architecture consisting of two layers. The composition core is interpreting SCaLE, thus operates on abstract services. A layer of execution agents (EA) accompanies the composition core.

An execution agent implements all functionality needed to execute services of a certain technology. For example, an EA for Web services will be able to receive, evaluate and send SOAP messages. Another EA is responsible for sending and receiving SIP messages.

At this point it is important to note that SCaLE does not define which EAs are available and how they are implemented. Also the API for attaching EAs is strictly not part of the language as such. However standardizing the EA API together with SCaLE makes most sense in order to allow coordinated design of exchangeable EAs.

All EAs use the same API towards the composition core. It is the main task of a EA to translate the technology specific details of the external interface into the general concepts of SCaLE. For example the reception of an external protocol message might be exposed to the core by means of issuing an event. When an external message is received; the composition core generates an event and issues this event to the respective composition sessions where further activities can be performed. These events also bear data received on external interfaces. This data might be mapped into shared state variables of the respective composition sessions when consuming the respective event.

This section uses an example based approach in plain English to describe the semantics of the key elements that constitute SCaLE. This choice is motivated by the graphical nature of the language and also by the target audience for this language that may not necessarily have background in Computer Science. However, a formal description of the proposed language using Structural Operational Semantics is available in Appendix B.

The proposed language follows the design principle proposed by Abelson et al. [157] and as such we emphasize on readability by non-expert designers. From this angle, the aspect that a composition core interprets this language is purely coincidental

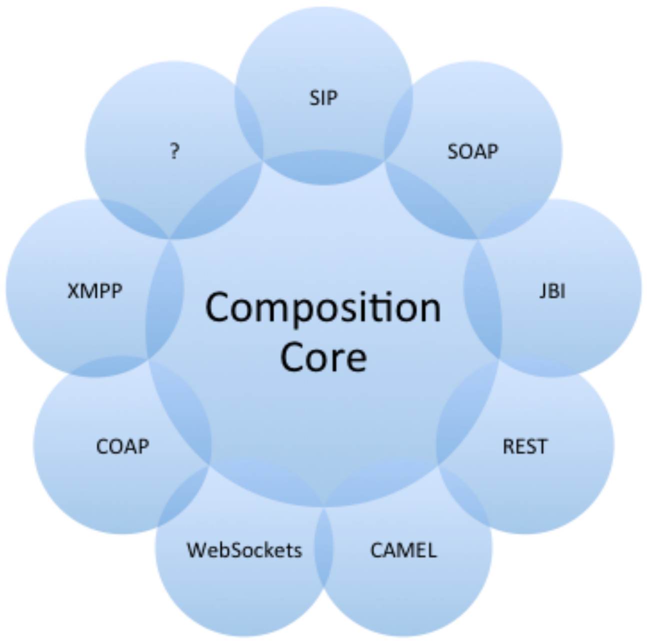

FIGURE 4.3: A technology agnostic core with modular technology support

and as such does not influence the language's design. The following constructs and aspects oft the proposed language are being detailed:

- application skeleton
- action

  atomic

  compound actions
- execution order
- action environment
- finishing an action
- conditional execution
- loops
- data type system
- annotations
- external type extensions

The description of the language is furthermore defined using operational semantics in Appendix B.

4.4.1 Application Skeleton

An application skeleton (or skeleton) is a collection of elements that aim at describing *what* the proposed composition does and not *how* it is done. That part is dealt by the underlying services. This particular process is described in more detail by the service template action.

In other words, a skeleton is the implementation of a composite service. More specifically, it describes which services should be executed and in which order. The definition of which service should be executed is done by means of a service action template, which contains a query that intends to select a constituent service. The predominant dependency between services is the order of execution, which is frequently implied by data dependencies, e.g. the output of one service is the prerequisite of another.

4.4.2 Action

An action is the most central element of a skeleton. It is used to define any kind of activity to be interpreted by the composition core. Figure 4.4 displays four actions. The graphical representation of an action is a rectangle. Actions are named elements. In the example shown in Figure 4.4 the action names are located in the top left corner of the rectangle. For purposes of simplicity, action properties are omitted in Figure 4.4, but they are later on described in 4.4.3.

An action may contain nested actions. Actions that contain nested actions are referred to as compound actions. Action Y is such an example. It contains the nested actions A and B. Action X has no nested actions and therefore it is an atomic action.

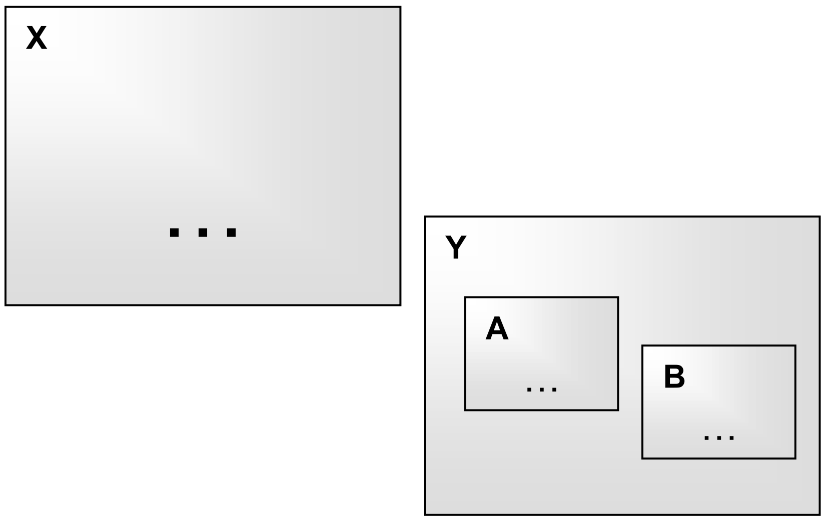

FIGURE 4.4: A skeleton with four actions

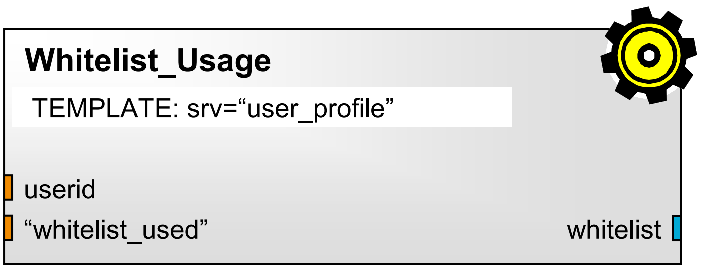

FIGURE 4.5: A service template action

4.4.2.1 Service Template — An atomic action

A service template is a specific kind of action; it serves as a placeholder that describes the properties of an external service to be executed. The composition core dynamically selects that service based on the query specified in the action.

Figure 4.5 portrays graphically a service template action. The keyword ”TEMPLATE” is used in order to specify, that this action is a service template. Following this keyword, a query is provided, formulated as a regular expression, describing the properties of the service to be selected at runtime by the composition core.

In this example the query used is srv=”user_profile”. This syntax denotes that the service to be selected should contain an srv property that evaluates true to the string ”user_profile”.

Input parameters for the service can be specified on the left side of the action. In this example, this action expects two input parameters. In a similar fashion, output parameters or effects that carry the result of service invocation can be defined on the right side of the action.

The service template action is a product of the ”command query separation” concept that was coined by Meyer et al. [115] and as such it is a combination of a query; returns a result (in this case a list of services) and does not change the observable state of the system, therefore it is free of side-effects, and a command; changes the state of the system.

4.4.2.2 Compound actions

A compound action is an action that contains nested actions. Figure 4.6 shows an example of such an action. In this example, action ”Whitelist_Preferences” consists of two nested actions, ”Translate_UID” and ”Whitelist_Usage”. ”Whitelist_Preference” is referred to as the parent action, while ”Translate_UID” and ”Whitelist_Usage” are referred to as children or nested actions.

When the composition core is called to interpret a compound action this results into the interpretation of the nested actions within. In this example, the nested actions are interpreted in a sequence; therefore ”Translate_UID” will not be interpreted until the end of the interpretation for ”Whitelist_Usage”. More information regarding the order in which actions are interpreted follows in section 4.4.4.

Another aspect in the context of compound actions is that of the locality of data. Within an action, either atomic, or compound, all variables are local copies. The composition core, prior to interpreting an action, creates snapshots of all variables that exist in the parent action. These snapshots are later on passed as copies to the nested action. The nested action is permitted to make changes to the variables copied from the parent but these changes will not be copied back to the parent action unless they are defined as the effects of the action.

In this example, the variable ”userid” is copied from ”Whitelist_Preference” to ”Translate_UID” since ”Translate_UID” requires this variable as input. Within ”Translate_UID” the variable ”id” is created in order to store the result produced by invoking the service ”translate_id”. This variable is an effect and as such it will be copied to the parent action, ”Whitelist_Preference”.

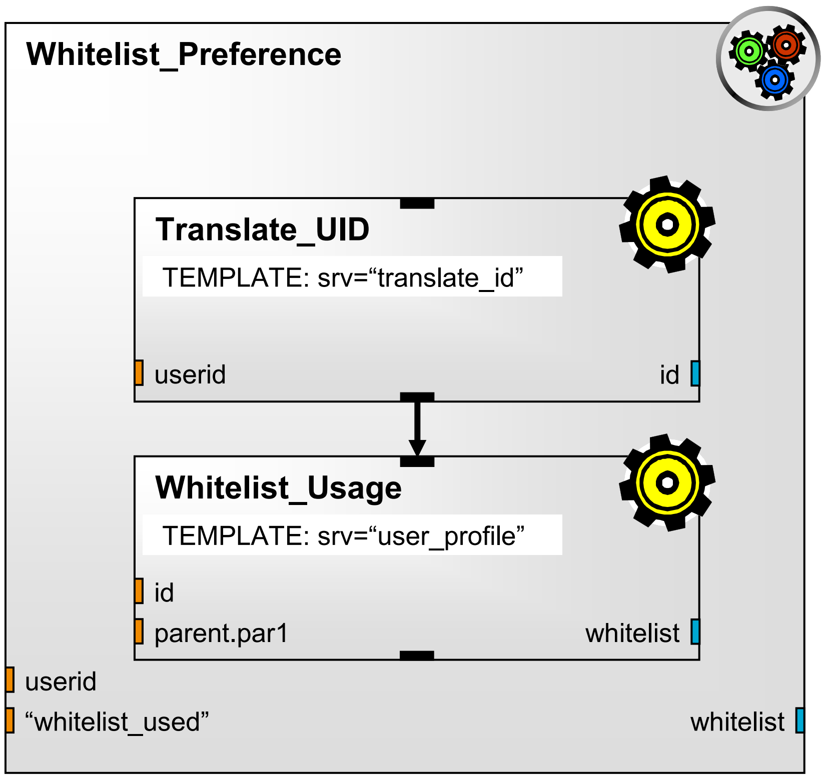

FIGURE 4.6: A compound action

In the case of the variable "whitelist", it will be propagated in the following manner; the effect of "Whitelist_Usage" leads to updating/creating the variable "whitelist" in its parent action, "Whitelist_Preference". The effect of "Whitelist_Preference" in turn leads to updating/creating the variable "whitelist" in the parent action.

4.4.3 Overview of Action

Figure 4.7 shows all possible properties of an action. Some elements such as action name are mandatory but most elements are optional.

Action name: Actions have a unique name. Other parts of the skeleton can refer to this action by using this name.

Start handle: Express graphically a starting constraint for the action. The action start is waiting at least until this handle receives a trigger, for example from another action's finishing handle. Using the start and finishing handles allows defining a static control flow manually.

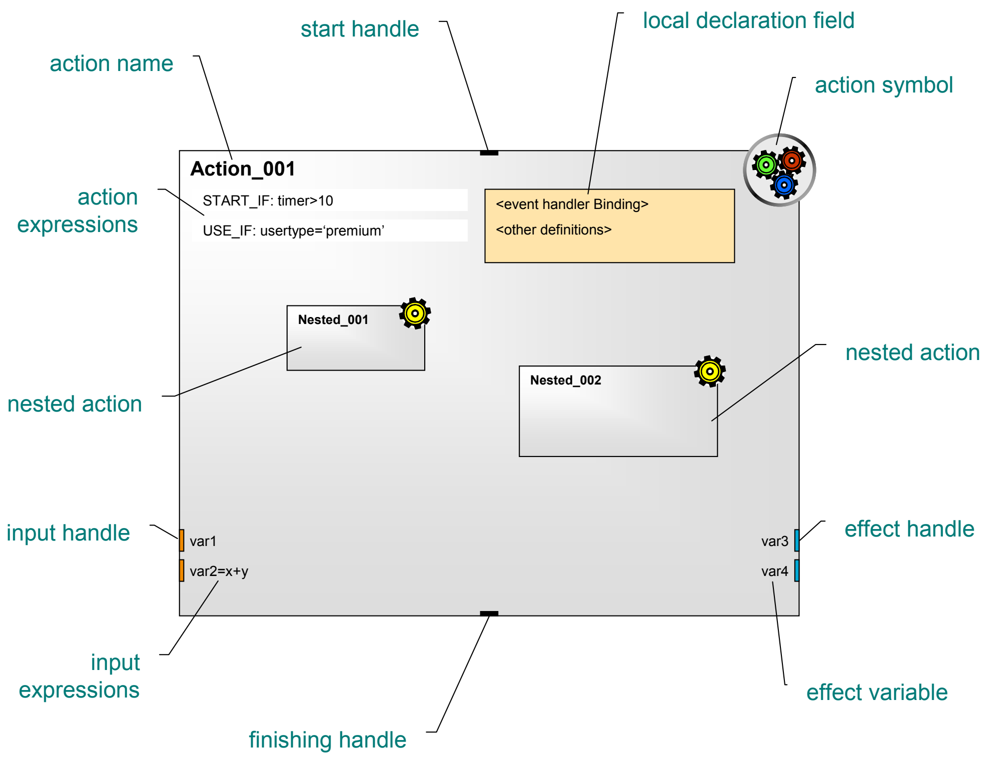

FIGURE 4.7: Action overview

Finishing handle: If the action finishes, this handle provides a trigger. This trigger can be used by means of drawing arrows in order to satisfy start constraints of other waiting actions.

Input handle: Input handles are used in order to define dedicated input parameters of the action. If the action is expressing a service template, these input parameters correspond to the parameters of the service that is about to be interpreted by the composition core.

Effect variable: All variables are local within an action by default. In order to propagate a variable to the parent scope, that variable has to be marked as an effect.

Action expressions: The action expression attributes specific semantics to an action. That is denoted by a keyword such as:

Template: Service template for query based selection and execution of services.

SELECT: Query based selection of services. Result is a list of services, which satisfy the query.

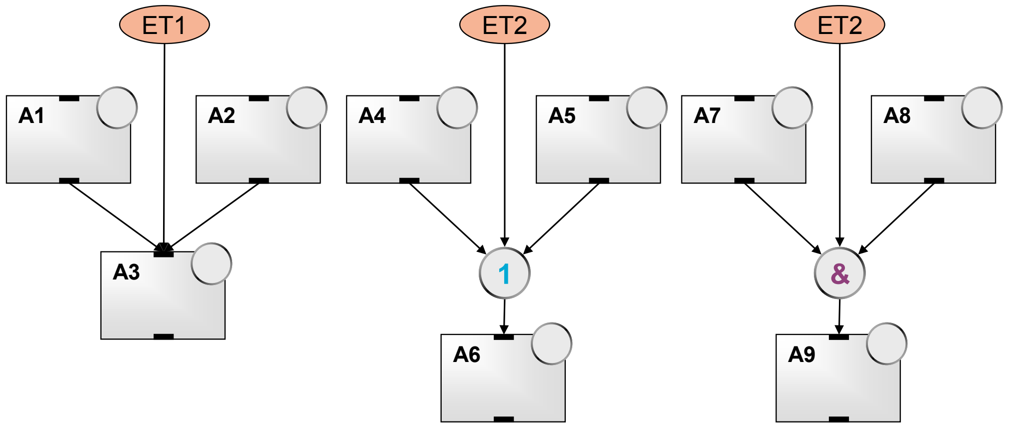

FIGURE 4.8: Execution order

INVOKE: Execute a service. Input is a set of services

DELEGATE: Delegate the execution to another action

SSM: Allows data manipulation such as assigning a value to a variable

CASE: Conditional execution with multiple alternatives

FOREACH: The action is executed for each element in a certain variable. This variable is usually a list. The iterations for each of the variable elements are executed subsequently

WHILE: Repeat an action as long as a condition is a true

Nested actions: An action can be compound, which means that it contains other actions. If there are no nested actions the action is atomic. Nested actions are isolated, thus they cannot be referred to outside from their parent.

4.4.4 Execution Order

At runtime, the composition core interprets actions. By default, execution order is influenced by data dependency. Each action can only be executed once the corresponding input, required by the action is made available. In the example shown in Figure 4.8, A3 can only be executed once A1's v1 effect is available. In the same fashion, actions that do not rely on any input can be executed in parallel. In Figure 4.8, A1, A2, A5 and A6, are such examples.

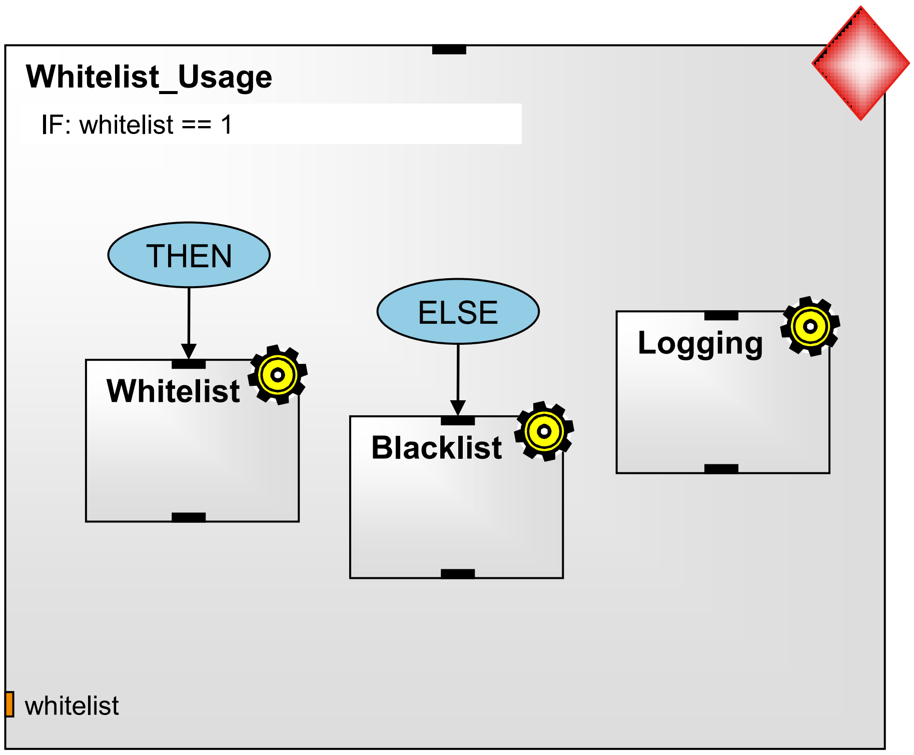

Figure 4.9: Execution flow

Default execution order can be overridden by using arrows. In such a way, one can enforce a sequential execution paradigm. Such an example is shown in Figure 4.8 where an arrow is placed between A1 and A4. This means that A4 can only be executed after A1 has finished.

If an action depends on multiple data dependencies originating from other actions certain expressions can be used in order to determine when the depended action should be executed.

- AND: An action will be triggered when all the actions that this action depends on have completed their execution.
- OR: An action will be triggered when at least one of the actions, this action depends on has completed it's execution.

An atomic action is considered to have completed its execution, when its atomic function, defined in the action expression has been executed. In the same fashion a compound action is considered to have completed its execution when all of its nested actions are finished.

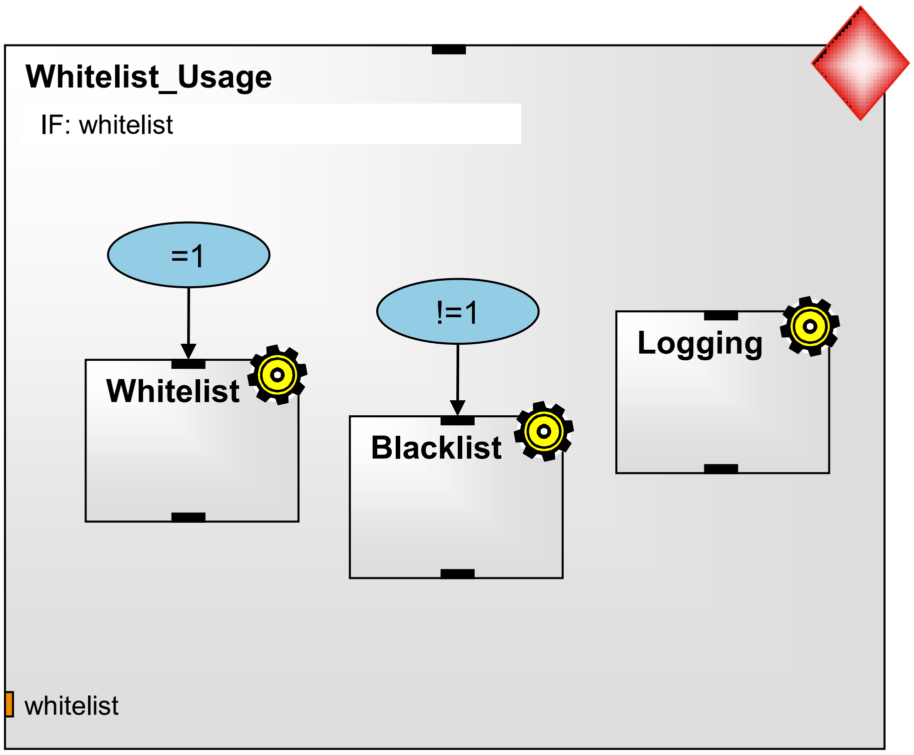

FIGURE 4.10: Conditional starting of nested actions

4.4.5 Action environment

The term action environment describes the locality of data within an action. In order to avoid race conditions all variables are local by default. Prior to the execution of an action, all variables of the action's parent are copied in order to create a new local environment. Once a local environment is created for an action, any changes made in any variable can affect only the local copies. An action's local environment can communicate with another action's environment by explicitly specifying a local variable as an effect. Variables marked as effects will be copied and therefore passed on to other variable's action environments.

4.4.6 Conditional Execution

Conditional actions are compound actions used to choose from different nested actions based on the evaluation of the conditional statement.

Figure 4.10 shows a simple conditional action. The action expression starts with the keyword "IF" followed by the condition. The condition is evaluated when the action

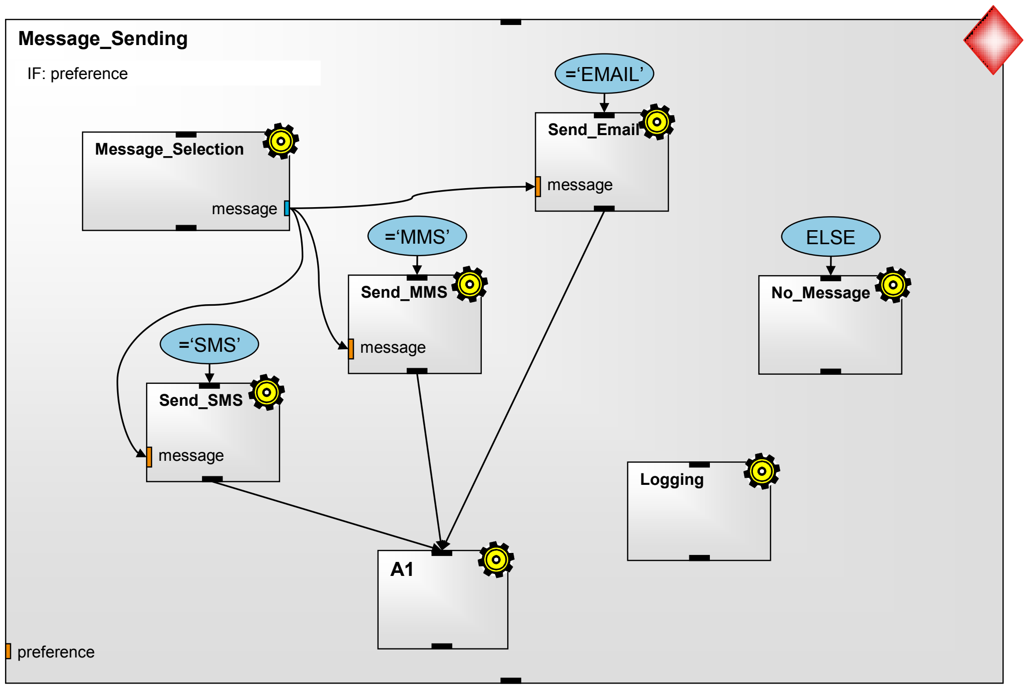

FIGURE 4.11: Alternative conditional starting of nested actions

starts. Labels are used to annotate the different choices that can be made such as THEN and ELSE. They correspond to the condition being evaluated as true or false. The ELSE label can be optional; if for example the condition evaluates to false and there is no ELSE label defined, then the action can finish.

In addition, Figure 4.10 contains a nested action called "Logging". This action is not connected anywhere. This means its execution does not depend on the condition. It is treated as a usual nested action and therefore it will be started in parallel with either "WhiteList" Or "BlackList".

An alternative usage of the IF type action expression is shown in Figure 4.11. The conditional expression can be placed on each corresponding label.

Next to an IF statement with two branches it is also possible to use a higher number of conditional branches. An example is shown in Figure 4.12. The condition is based on evaluating the variable "preference". In this example the conditions distinguish, if the variable 'preferences' contains the strings 'SMS', 'MMS', or 'EMAIL' in order to send a notification to a user in the preferred way.

If none of the conditions is fulfilled, the branch marked with ELSE is triggered. The action called 'No' is always triggered in parallel to the other actions. It is not

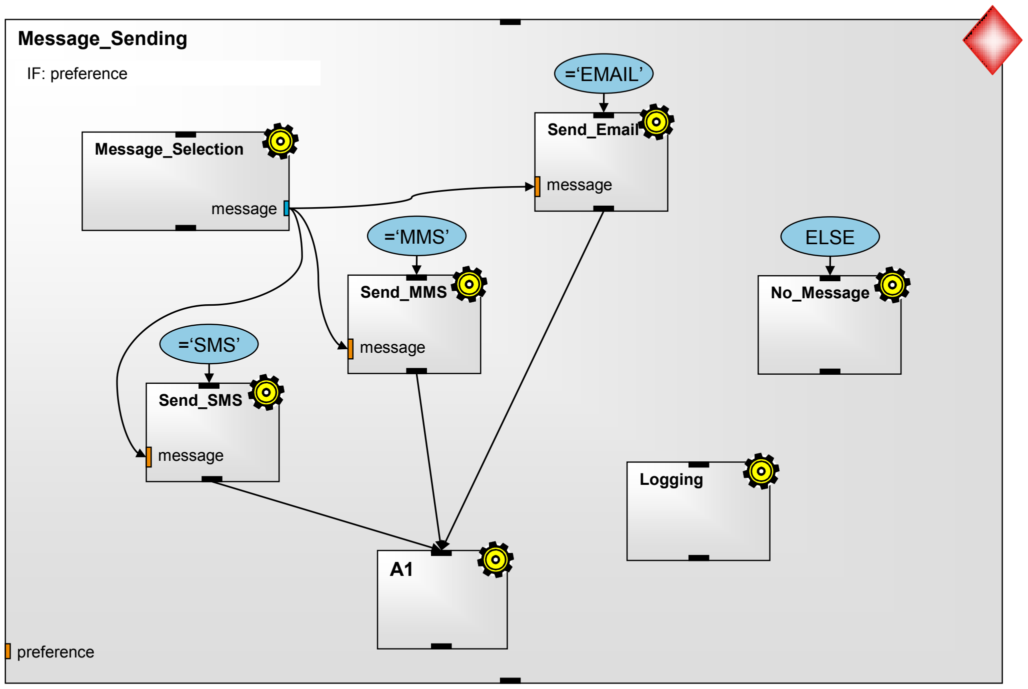

FIGURE 4.12: High number of conditional branches

part of the conditional triggering but depends on its parent's start. The action 'A1' is triggered from all three options. In this example, the message sending service 'Send_SMS', 'Send_MMS' and 'Send_EMAIL' depend on the actual message text to be available. The text is generated by the service 'Message_selection'. In this example, this is modeled by means of a data dependency. The IF condition is evaluated in the beginning of the compound action 'Message_Sending'. The result of the evaluation triggers the corresponding alternative. Due to the data dependency, the actual start of the triggered message sending service is postponed until the service 'Message_Selection' has issued a message text.

The condition expression can also be completely moved to the trigger handlers and it can consist of a complex expression based on multiple variables and complex operators.

4.4.7 Loops

Loops are created by means of compound actions. The action's content, including its nested actions is what is executed at each of the iteration of a loop. Two types of loops are supported. WHILE loops iterate the compound action while a condition

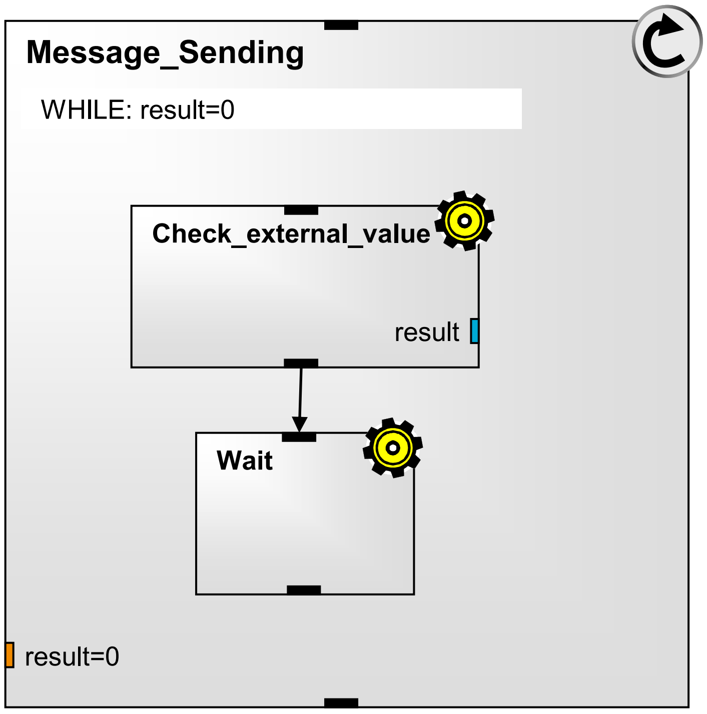

Figure 4.13: WHILE loop

is true. FOREACH iterates through all elements of a list or map and executes the compound action again for each of these elements.

An example of a WHILE loop is shown in Figure 4.13. The nested actions are repeatedly triggered and executed while the condition is fulfilled. Here the loop is repeated again if the variable is set to 0. This example implements polling of an external value until it is not 0 any more. The action 'Wait' leads to a certain delay until the next loop is run, thus the polling of the next value is done.

The FOREACH action expression creates a loop through all contents of a list or a map. An example is shown in Figure 4.14. The variable 'userlist' contains a list of usernames. The loop will iterate through the entire list by triggering all nested actions again for each of the users. The variable 'user' contains the list element for which the iteration is done. In tis example a Message will be sent to all users in the list. Note that a FOREACH loop handles all iterations sequentially one after the other.

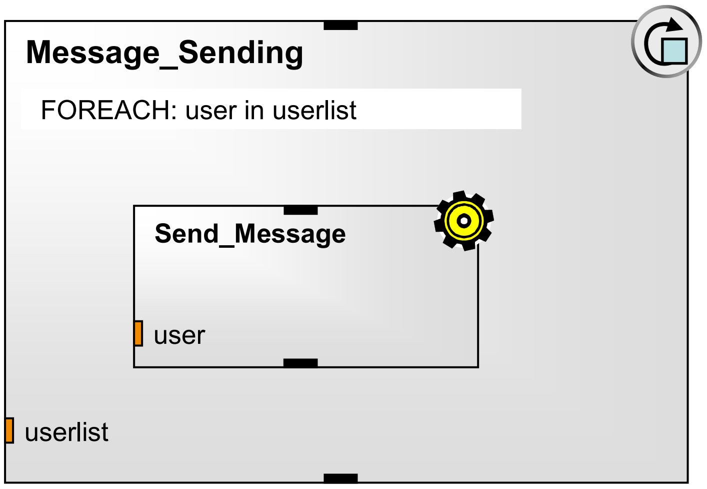

FIGURE 4.14: FOREACH loop

4.4.8 Events

We consider the following sources of events:

- Reception of an incoming protocol message,
- Events explicitly issued from actions
- Activities on the local environments like an update to a shared state variable
- Session related activity like start and end of sessions
- Execution details, like start and finishing of actions
- Errors and exceptions
- Timers

The interaction of a composition session with external services is relying on events to a great extent. A SIP message or a SOAP message are exposed towards a composition session by means of events

4.4.8.1 Basic Event consumption

An event can be directly consumed within a skeleton. An event trigger point is used for this purpose.

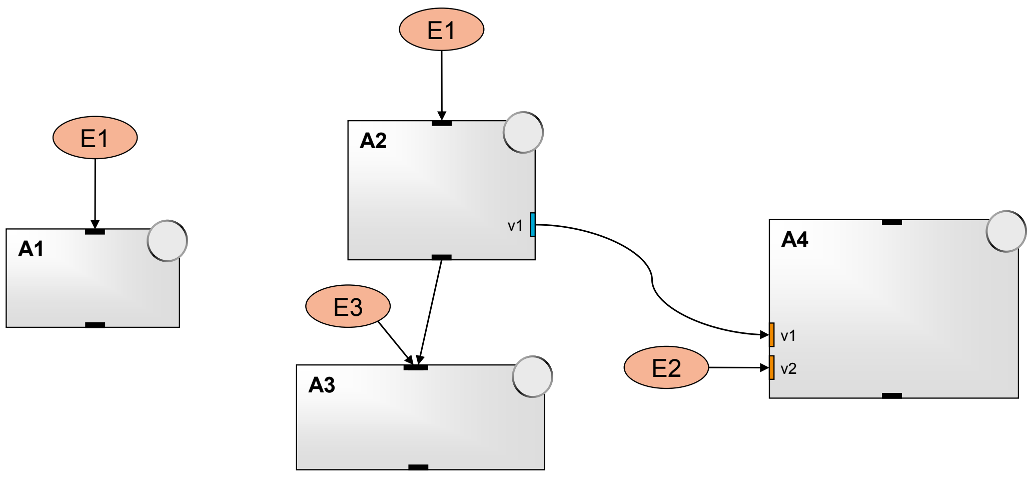

FIGURE 4.15: Consuming events

The example in Figure 4.15 shows some possibilities to use events in skeletons and together with actions. There are three event trigger points for the respective events E1, E2 and E3. This event trigger point issues and triggers once the event occurs. This trigger can be consumed as usual by actions. E1 for example is connected to the start handle of the action A1. This means the action A1 is started once E1 occurs. Actually, a new instance of A1 is started each time E1 occurs, because each time a new trigger is issued.

In this example the same event E1 is also used to trigger A2. Thus, once E1 occurs, both triggers are issued. Instances of both, A1 and A2 are started and with A2 also the subsequent actions A3 and A4 might be started.

Please note, that A4 as it is defined in this example is started with its parent (here the skeleton) rather than explicitly by an event. This means that here only one instance of A4 is started regardless of how many events are triggered. Furthermore, A4 has input dependencies. It waits for the variable v1 to be written by A2. This implicitly means that A4 can only start after A2 starts and updates v1. A2 in turn is waiting for the event E1.

Another event E2 is connected to the second input of A4. This expresses that the actual start of A4 is deferred until E2 occurs and issues a trigger to the input point for v2. This kind of event consumption is helpful in case E2 is an event related to an update of the variable v2.

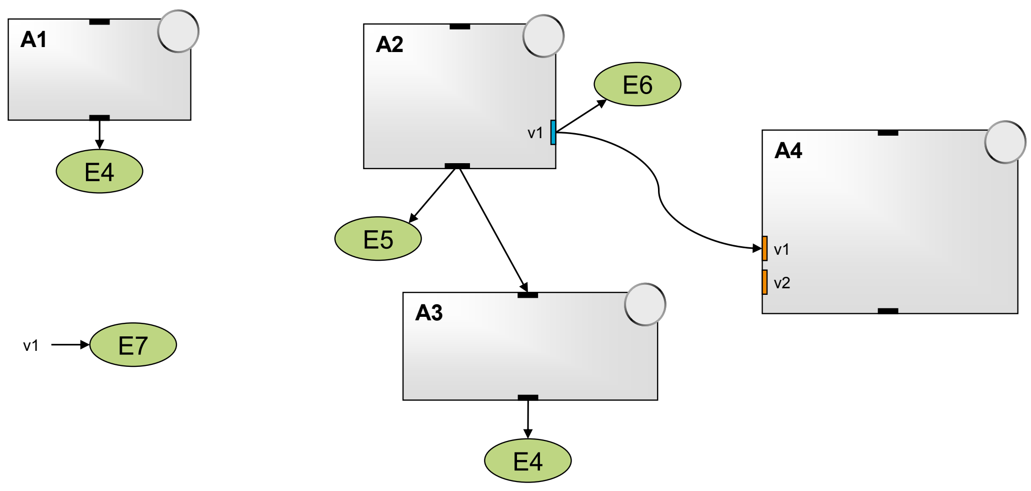

Figure 4.16: Issuing explicit events

The event E3 shows, that multiple trigger sources can be used towards a single action start handle and that event trigger points can be added to other kinds of trigger sources.

4.4.8.2 Explicit Event Generation

Events can be issued explicitly from the skeleton elements. Linking triggers to events achieve this.

The example in Figure 4.16 shows some of the possibilities to specify explicitly in a skeleton that an event shall be issued. The event E4 is issued once the finishing handle of A1 or A3 issues a trigger. If A2 is finished, this triggers not only A3 through the start handle, but also the event E5 is issued.

If the variable v1 is updated through the effect of A2, also the event E6 is issued. So also for the event generation the data related triggers could be used. This is also shown for E7. Once variable v1 is created or updated the event E7 is issued.

Explicit events are therefore all events designed by the skeleton developer.

4.4.8.3 Generation of External Events

Next to explicit events generated directly from skeletons and actions, events issued by the composition execution engine are of high importance. External protocol messages will be exposed as events towards a composition session. This allows designing dynamic reaction on external activity. This also means that even the initial external trigger, which caused the composition to start, is exposed as event.

The respective execution agent determines the event name of external events. Policies of the engine might also be used in order to determine globally which events are issued to composition sessions at all.

4.4.8.4 Generation of System Events

Activities of the composition execution engine are another source of events. For example errors and exceptions are exposed like events.

4.4.8.5 Data conveyed in events

Events can have data associated to them. For example the Message parameters of an external protocol message are conveyed within events and exposed to the handling session as event data.

This data is loaded into the local environment of the event handler when the event is consumed.

4.4.8.6 Event Locality and Propagation

In order to achieve a controlled event handling it is important to define rules, to which actions events are issued first and how they are propagated.

SCaLE distinguishes local and non-local events. Local events can clearly be associated to the local environment of a skeleton or action. It is that environment where the event was issued from. Explicit events are a good example of a local event, but also system events can be local, if their root cause is within a local environment. The propagation rules for local and non-local events are different.

Local events clearly originate in the execution of a particular local action, thus are considered to be local to this action's environment. By default such local events are consumed locally and they are not propagated to this action's parent or even globally. However, such propagation of the local event can explicitly be initiated. Local events can be propagated to the parent and they can also be issued on global level. Exposing and propagating the event to the parent is an effect of the action and it is also shown as an effect. The event becomes a local event in the parent environment. Issuing the event on global level leads to treating it as non-local as described below.

Non-local events are system events and external events that cannot be assigned to a particular local environment. A policy defined on global skeleton level determines how a particular non-local event shall be treated and propagated through the skeleton. The default is the following handling: Non-local events are propagated through all actions in a certain order that is derived from the nesting of actions. The general rule is that non-local events are propagated from inside (lowest nested level) out (global level).

To understand the propagation principle it is important to understand the tree like nature of the action-nesting stack. Action nesting and parallel execution together will lead to multiple nesting stacks in parallel, which are also dynamically changing throughout the composition execution. As a result the nesting of actions has a tree like structure with the global skeleton level as root of the tree and the lowest nested level as leaves. An example is shown in Figure 4.17. The example shows a skeleton with nested actions and the respective nesting tree.

If there are multiple nested branches, the propagation of non-local events starts independently at all leaves. If a non-local event is issued to this composition session, the event is first offered to all these actions, which are the endpoints of the nesting tree.

If a non-local event is issued to an action, which has no event handling defined for it, the event is automatically propagated in the nesting tree branch and issued to the parent.

If a non-local event is issued to an action and the action consumes it. This eventually stops the event propagation within the nesting branch of the action. The event is not automatically propagated unless the action explicitly propagates it to its parent. This means, from an action's point of view the event handling for local and non-local events is similar in the respect, which propagation to its parent needs to be explicitly

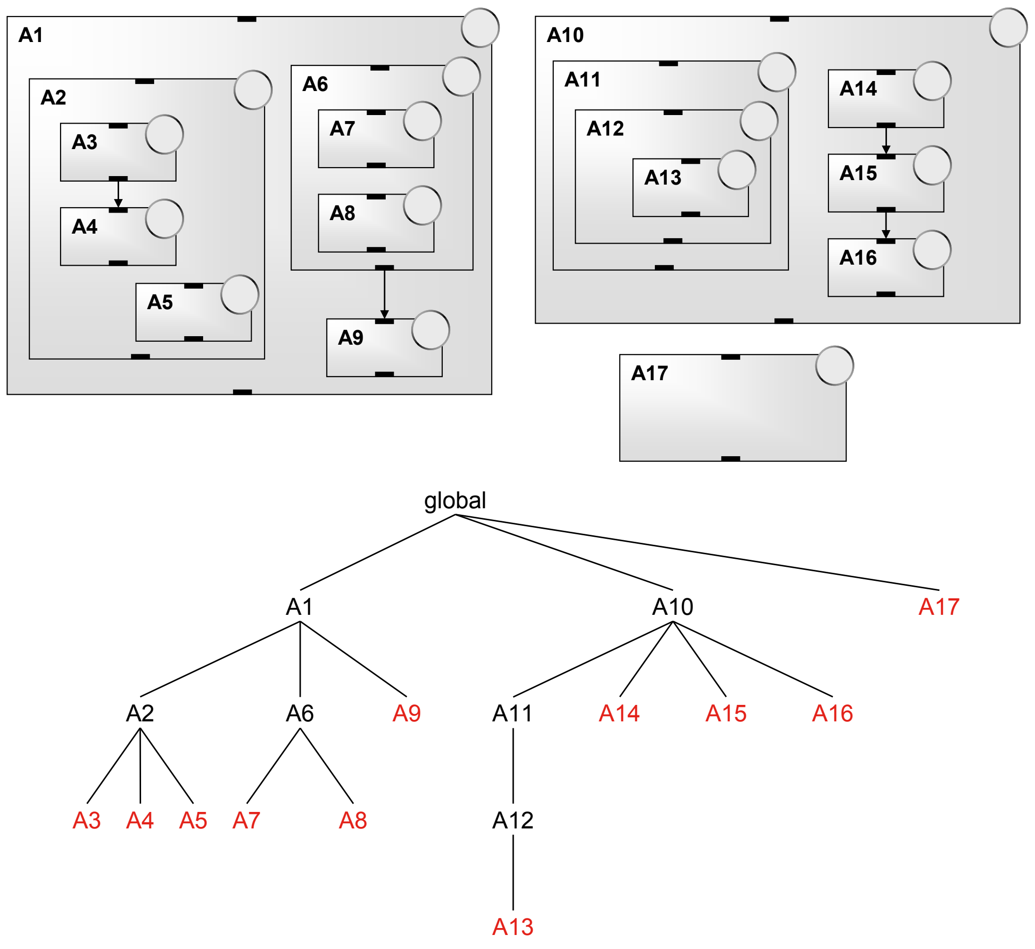

FIGURE 4.17: Action Nesting Stack

initiated. Please note that consuming an event can stop further propagation within one nesting branch, but does not effect event propagation in other branches of the nesting tree.

At the points where two sub-branches join, the event might be received from both sub-branches. This means an action can receive the same event instance several times, because it was propagated from several of its nested actions. It is up to the action to decide if subsequent receptions of the same event instance will also be handled multiple times or ignored.

Event propagation through the entire nesting tree appears to be an enormous and time-consuming effort if the nesting tree is big. However, it can be expected, that only a minority of actions define event handlers for a particular event and only these actions need to be taken into account. The actual propagation tree for a particular event is therefore much smaller than the entire nesting tree.

The event propagation policy on global level can deviate from the described default event propagation method. For a certain event it is possible to determine that it is only issued on global level and not propagated through all nested actions. It is also possible to determine that an event is always also issued on global level. This ensures that global event handling can consume it even if the event was already consumed by all nested actions. This policy is valid per skeleton, thus the same external event, if it is issued to several composition sessions might be treated differently from session to session.

4.4.8.7 Event handlers

Event handlers are dynamically composed out of actions. This means, an event is handled by issuing a starting trigger to the starting handle of actions. This triggers action execution and can cause a number of subsequent actions and nested actions to also start. Which actions are started in order to execute the event handler is decided by constraints as usual. The composition engine therefore also composes the event handler dynamically. Therefore there is usually no clearly unique and distinct entity, which can be called to be the event handler for a particular event. A number of actions being executed based on a trigger issued, because of an event was received, constitutes the handler.

The event itself is consumed in the skeleton by means of triggers issued through event trigger points. If the event occurs several times, for each event a new trigger is issued, which in turn will trigger a new instance of the connected action. Therefore, with each event also a new composition of an event handler is started.

4.4.8.8 Event Filtering / Event Binding

Event binding is a very powerful way to consume and filter events before they are further consumed and handled by means of for example trigger senders. This chapter outlines some of the possibilities.

An event binding is a filter for events. The binding is invoked by the reception of an event. It contains an expression, which formulates a condition and as result it allows specifying what shall be done. Figure 4.18 illustrates this principle.

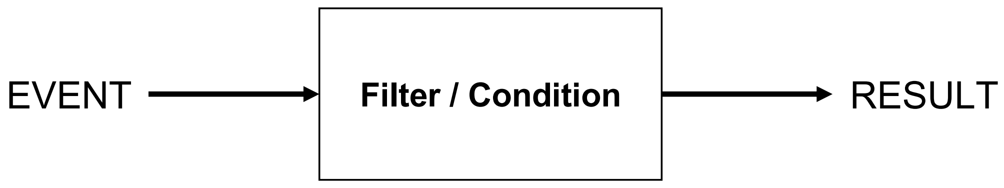

FIGURE 4.18: Event binding

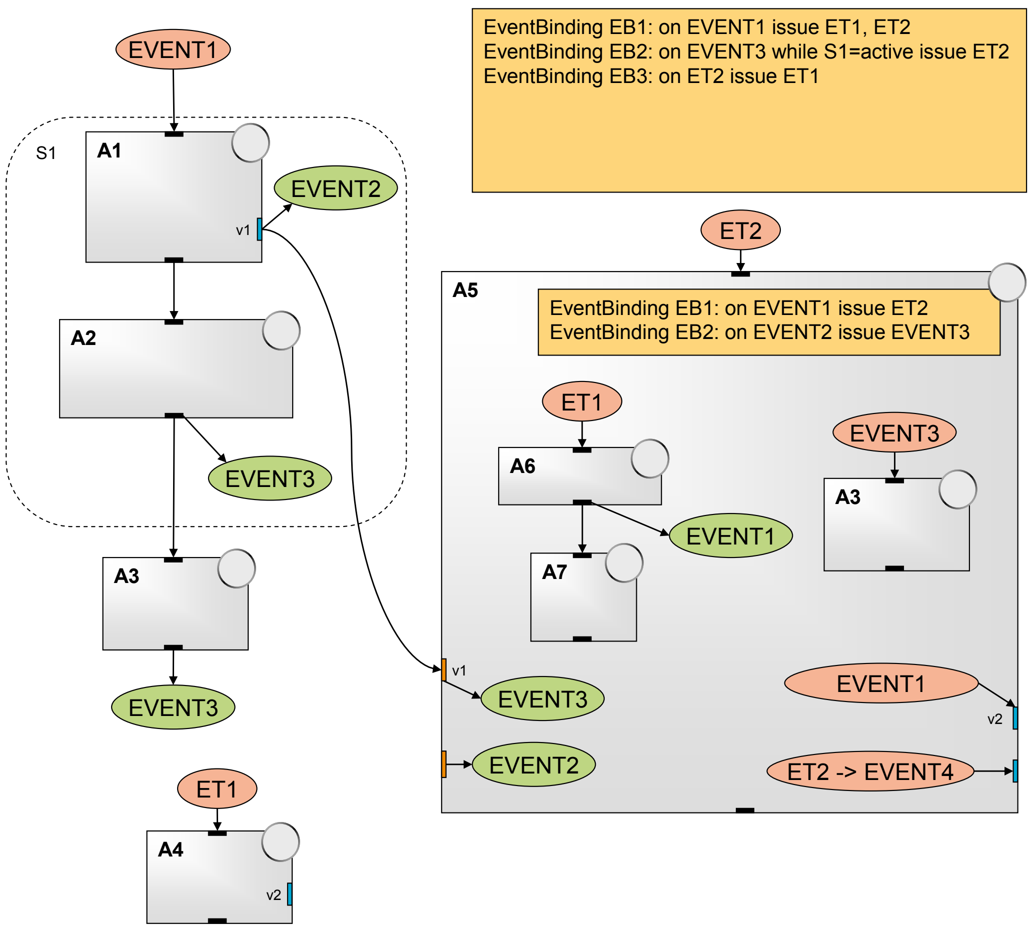
EventBinding EB1: on EVENT1 issue ET1, ET2
EventBinding EB2: on EVENT3 while S1=active issue ET2
EventBinding EB3: on ET2 issue ET1
EVENT1
S1
A1
EVENT2
v1
A2
EVENT3
A3
EVENT3
ET1
A4
v2
ET2
A5
EventBinding EB1: on EVENT1 issue ET2
EventBinding EB2: on EVENT2 issue EVENT3
ET1
A6
EVENT1
A7
EVENT3
A3
v1
EVENT3
EVENT2
EVENT1
v2
ET2 -> EVENT4

FIGURE 4.19: Events and bindings

Event bindings are defined within the action's local declaration field, thus the event binding definitions are local. Event bindings are named and can therefore be referenced for applying changes dynamically.

An event binding can map input events of any type event to an explicit output event within the local environment. This mapping can be conditional, thus the output event is only thrown if next to the reception of the input event also a condition is met. Scopes can be used here. An output event is for example only issued if a certain scope is active.

Figure 4.19 shows some possibilities. The event EVENT1 is consumed directly as trigger for the action A1. Additionally through event binding EB1, EVENT1 is mapped to ET1 and ET2. The event binding EB1 issues ET1 and ET2 each time EVENT1 occurs.

The event EVENT3 is issued if actions A2 or A3 finish. The eventbinding EB2 issues ET2 if EVENT3 occurs while scope S1 is active.

Event handling is by default tied to the local environment. The event EVENT1 within the compound action A5 is for example a different event than EVENT1 in the parent environment.

Within A5 there are some examples of events related to data input and effects. EVENT3 within A5 is for example issued each time a trigger is received related to the input of variable v1.

EVENT2 within A5 is referring to the event of the same name in the parent environment. This means each time EVENT2 occurs in the parent, it also occurs within A5.

EVENT1 within A5 is triggering the effect related to variable v2. While the default behavior of the effect is to trigger each time the variable is updated, this effect only triggers if EVENT1 occurs. This can be used for controlling the effects explicitly.

The second effect of A5 is triggered in case of ET2 occurs. This effect issues EVENT4 into the parent environment. This means, that the occurrence of ET2 within A5 is exposed as EVENT4 to the parent.

4.4.9 Data type system

In SCaLE variables can be used without explicit typing and without declarations. Variables are created on demand and composition core selects the type, which is fitting best to the data only when this is required (latent typing). In addition, the composition core performs data conversion if the variable is used in a way that demands a certain type. For example, the composition core automatically tries to convert a value into a numeric value once this variable is used within an arithmetic operation.

This way of handling data types is very intuitive in most of the cases. However, SCaLE allows using explicit typing by means of annotations. The data type annotation is identified by the keyword ”TYPE”. It is the only annotation that is by default configured and available in the composition core. Nevertheless it is optional.

Basic data types supported by default are String, Integer, Real and Boolean. Complex data types supported by default are List and Map. A list is a set of variables referenced by an integer index. A Map is a set of variables referenced by a key value.

4.4.10 Annotations

Annotations are optional properties of data elements like for example a variable or the input and effects slots of actions.

Annotations are specified in a key value pair fashion. The key is the type of property and the value is the actual property of the data element.

Annotations are mutli-dimensional. This means several properties of data can be assigned independently to a single data element. This also means that the annotation is a list of independent key value pairs.

A simple syntax for a variable with assigned annotations is:

```
Var1(TYPE=string; FORMAT=email\_address] = alice@net.com
```

The previous example shows a two dimensional annotation consisting of a type definition and a property describing the format. So the email address of user Alice is stored in variable var1. The annotation is used to define that var1 contains a string and that the string has the format of an email address.

When executing an operation with the variable the way the annotations are handled is not specified by SCaLE. SCaLE only allows annotating data elements. The functionality using annotations is externally defined and is optional. This means that functionality to set remove and otherwise manipulate the annotations is within an optional external component. The core engine will ignore them. The only exception to this is data types as described in the following sub section.

4.4.11 External Type Extensions

Registration of new types and respective handlers for type related operations are possible through a dedicated API of the SCaLE engine. With a new type also the implementation of certain type conversion routines and operates needs to be registered.

A new type may be needed in order to define data handling that is specific to a service technology. This is in particular the case if complex data constructs are needed. Their details might be lost when storing them in variables only relying on the basic data types of the engine. As these complex types are specific to the service technology, they are defined external of the core engine.

One example where external types may be needed is the storage of a SIP message. It is structured in parameters, which in turn can also have a complex structure.

Another example is SOAP based Web Services. Their data exchange relies on XML and thus can have a complex structure.

Both examples show technology dependent specific data that shall therefore not be integrated into the composition core. Having additional types defined in optional external modules means that the stays future proof in two respects; In case a new service technology evolves, their specific data handling can be added in a modular way. Furthermore, no data type specific to an old service technology stays in the core engine once this technology is removed thus leaving the system clean of legacy.

4.5 Examples

One of the main design guidelines behind SCaLE is that the underlying technological details of a service are only instantiated through the process of service binding and invocation. There are three fundamental ways of interacting with external services:

- Synchronous request-response: A request is sent to the service and the user waits until a result is delivered
- Asynchronous message exchange: A service is a process, which is executed asynchronously to the client process

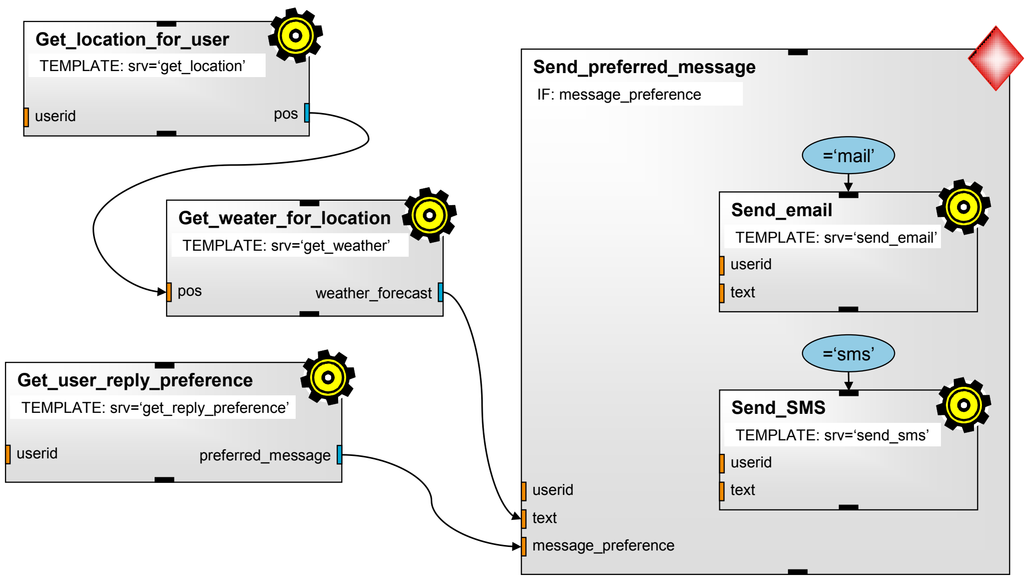

FIGURE 4.20: Weather information

- Service chaining: The service is a self-contained entity allocated on a message chain that might ne part of an end-to-end session.

This section provides three examples that detail how the composition core deals with these distinct interaction patterns with external services. At the end of this section we provide an additional example that describes a heterogeneous example that combines all three interaction patterns.

4.5.1 Synchronous request-response

Re-using external services through a request-response interaction pattern is the most simple example because the interaction between the user and service is based on a single request being sent to the service and a single response being returned. Re-using SOAP based web services usually follows this pattern. That said, in some cases the return response can be omitted, either because the external service may not provide one, or because the caller making the request is not interested in the response (request-and-forget).

Figure 4.20 shows an example of a simple weather information service based on SOAP that follows the request-response interaction pattern. The action "Get_location_for_user" selects and invokes a service that will provide the geographical location of the user.

As the selected service is a web service being consumed in a request-response fashion and synchronously, the action finishes after the location was received as the service's response.

The other service template type actions in this example operate similarly. The action ”Get_user_reply_preferences” accesses the user profile in order to determine the preferred way to send an answer to the user. ”Get_weather_for_location” provides a weather forecast for a specific geographical location. The compound action ”Send_prefered_message” is waiting for the message preference information and the text to be sent, before it decides whether to send an email or a short message.

4.5.2 Asynchronous message exchange

If the goal of the skeleton is to communicate with another independent process, this communication will consist of multiple incoming and outgoing messages that are asynchronous. Message dispatching will be done through service template type actions. However, this type of service invocation does not necessarily need to wait for a response.

Responses will be asynchronously received messages. They are exposed towards the session as data dependencies. In order to co-ordinate a state machine is modeled.

The example in Figure 4.21 implements a skeleton that subscribes to a stock exchange information system in order to get information of a particular share. It is assumed that there is a service, which can regularly send updates of the share price. This example subscribes to this service with the share the user is interested in.

4.5.3 Service Chaining

Although the roles and topology when using SIP services differs significantly from Web services, SIP services are selected based on queries and service template actions. From the designer's point of view, first of all it is important to take the decision and select which service to put on the SIP service chain. This task is mainly guided by selection actions and their corresponding selection queries.

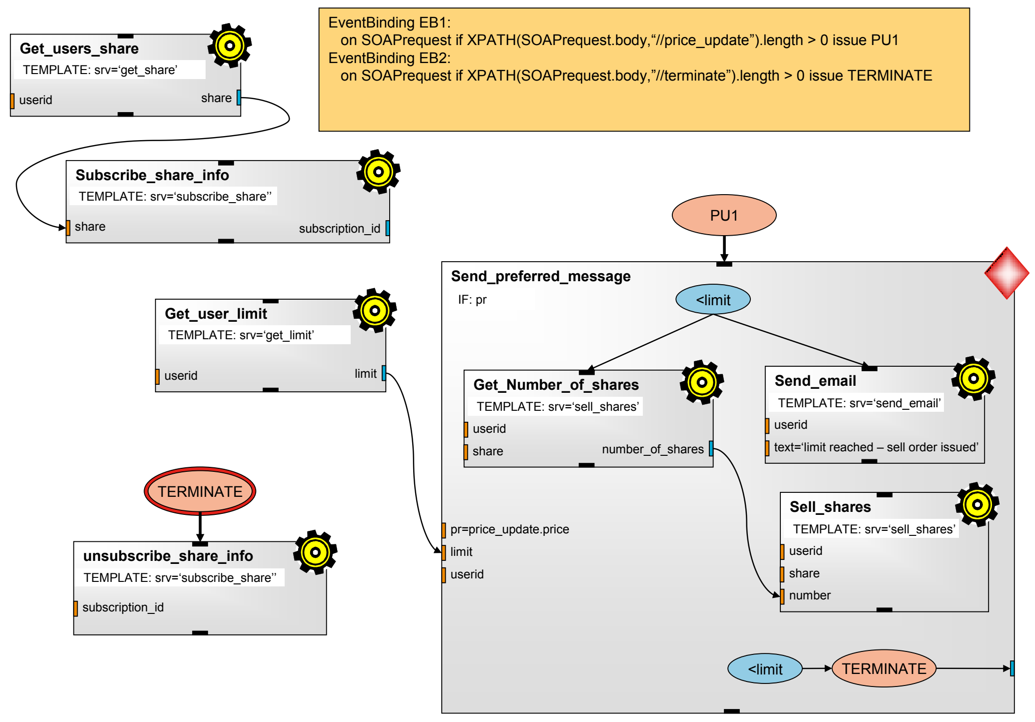
EventBinding EB1:
on SOAPrequest if XPATH(SOAPrequest.body,"//price_update").length > 0 issue PU1
EventBinding EB2:
on SOAPrequest if XPATH(SOAPrequest.body,"//terminate").length > 0 issue TERMINATE
Get_users_share
TEMPLATE: srv='get_share'
userid
share
Subscribe_share_info
TEMPLATE: srv='subscribe_share''
share
subscription_id
Get_user_limit
TEMPLATE: srv='get_limit'
userid
limit
TERMINATE
unsubscribe_share_info
TEMPLATE: srv='subscribe_share''
subscription_id
PU1
Send_preferred_message
IF: pr
<limit
Get_Number_of_shares
TEMPLATE: srv='sell_shares'
userid
share
number_of_shares
Send_email
TEMPLATE: srv='send_email'
userid
text='limit reached – sell order issued'
Sell_shares
TEMPLATE: srv='sell_shares'
userid
share
number
pr=price_update.price
limit
userid
<limit
TERMINATE

FIGURE 4.21: Stock Exchange Monitoring

Once being on the SIP service chain, the service is a self-contained unit taking its own decisions based on the end-to-end signaling it is listening to. Crucial in such a topology is the order in which services are selected in order to build the service chain. The second decision of the designer is therefore where the service shall be allocated. In practice, within the dynamic process of service chain building this "Where" decision regarding the service order is actually a "When" decision. The chain is build successively one service after another. Thus, this part of the designer's decision is subject to queries that are guiding timing and application skeleton execution order.

Two or more services without any direct or indirect dependency regarding their execution order can be specified in SCaLE. They are considered to be independent of each other. If these services are instantiated as SIP services, no clear order for these services to be allocated in the SIP service chain can be derived from the application skeleton in order to guide the designer. However, SIP service invocations need to be serialized in order to create an ordered SIP service chain. The composition core will place SIP services onto the SIP service chain in the order in which they are selected. If there are no ordering constraints, any resulting order, thus any resulting service sequence on the SIP service chain is possible. This means, if the order in which the services reside on the SIP service chain matters, additional queries are needed to express this dependency. It is often the case that the order of services is crucial in SIP. The same services put into a different sequence will usually result in a significantly different user experience.

The example shown in Figure 4.22 provides a composition consisting of three services. Here explicit ordering is used to express, that the execution order shall be Service 1 before Service 2 before Service 3. The composition is triggered by the event sip_invite, which is bound to the reception of the SIP INVITE method from an IMS network. A SIP INVITE request is a typical first forward message in order to establish a communication session with a user.

In this example the application skeleton is used in order to compose three services at reception of a SIP INVITE. Here, for the purposes of simplicity, all three services will be instantiated as SIP services. A heterogeneous example follows later on in subsection 4.5.4. Figure 4.23 shows a sequence diagram between the users, the IMS, the composition core, and the three SIP services at establishment of the end-to-end SIP session with the two users BOB and ALICE as the endpoints.

On reception of the INVITE message from BOB, the SIP Application Server (SIP AS) contacts the composition core in order to retrieve services to be placed in the end-to-end session. The composition core will trigger the application skeleton shown in Figure 4.22.

According to the application skeleton, the composition core will start building the composition by selecting service 1. SIP AS will send the SIP INVITE to the selected service 1, where an instance is started. Service 1 returns the SIP invite to the SIP AS and is now logically allocated in the SIP service chain. SIP AS will then ask the composition core, if there are more services to consider. The composition core will then proceed similarly with services 2 and 3.

Please note, that from the point of view of the composition core, the service template action for service 1 is not finished before the SIP AS reports about the status of the SIP service execution. Usually this coincides with asking the composition core for the next service to add. This request for more services might implicitly confirm that the previous task of adding a service was done.

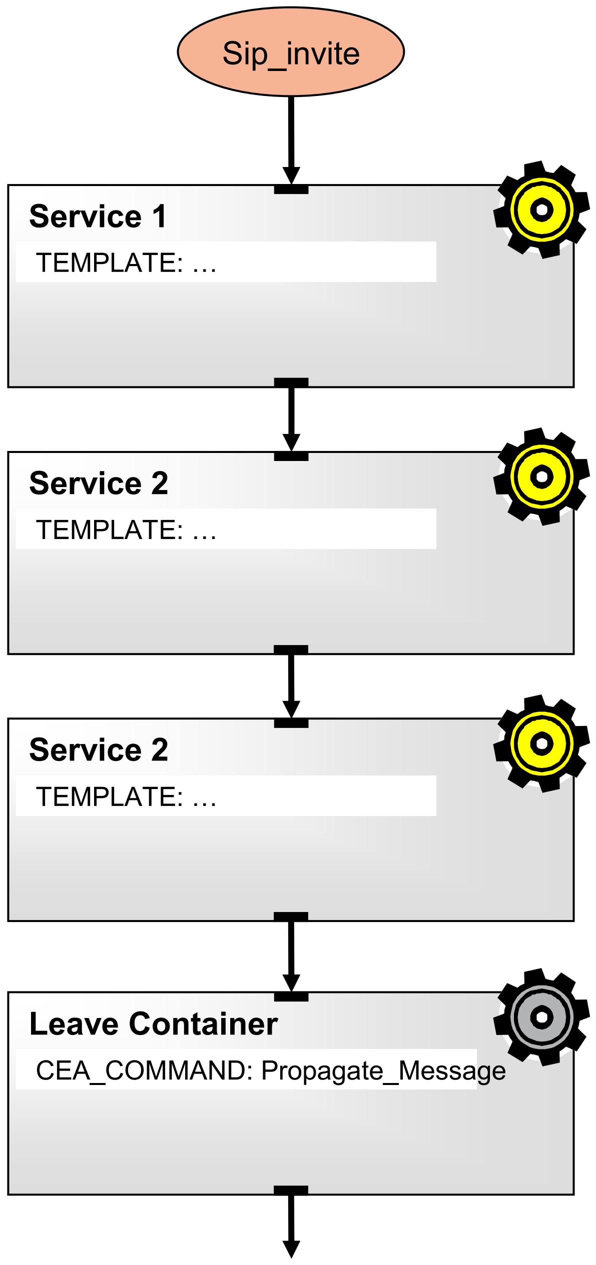

FIGURE 4.22: Skeleton resulting in a SIP service chain

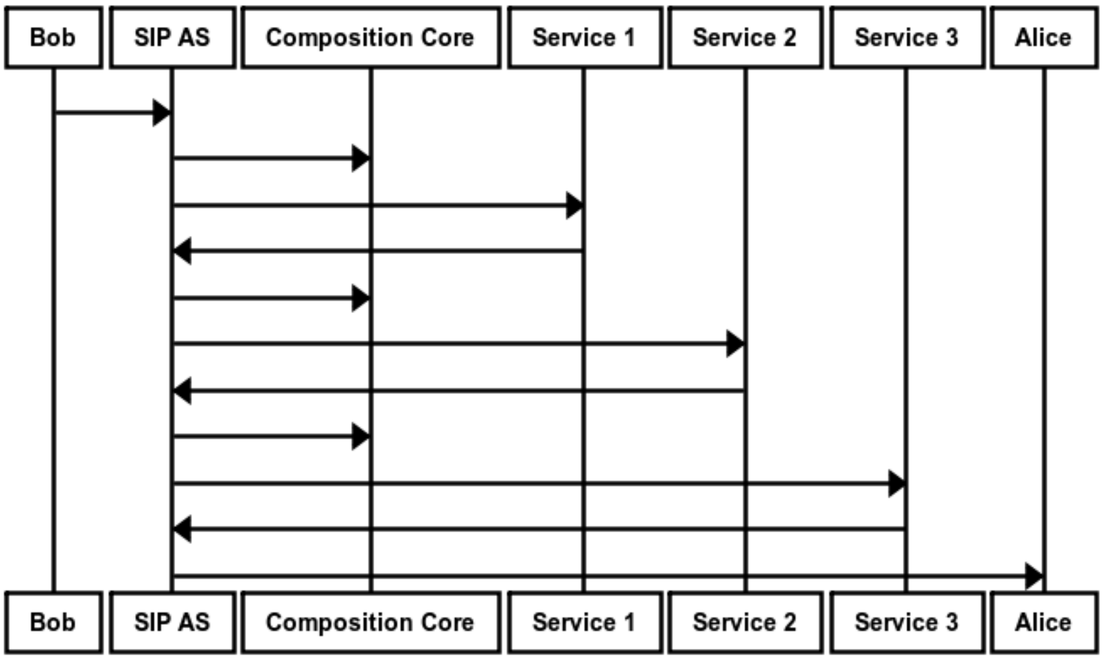

FIGURE 4.23: Communication sequence including users, SIP services, and composer

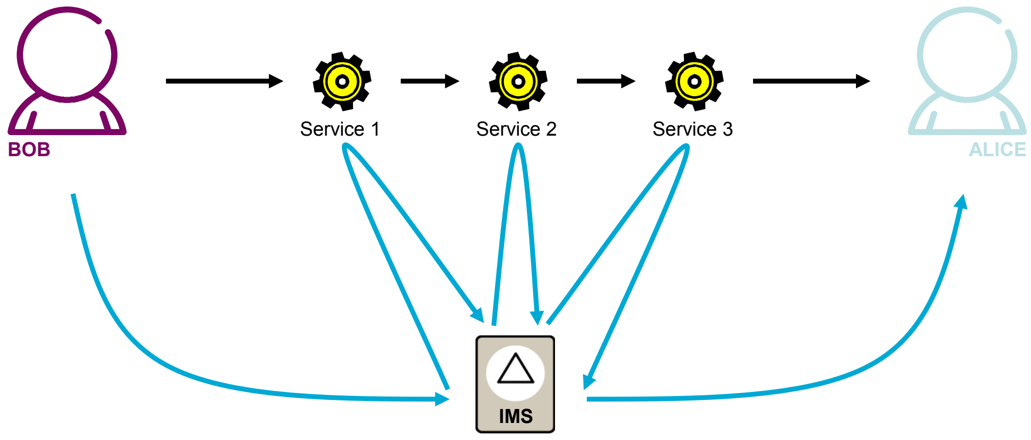

FIGURE 4.24: The resulting SIP service chain

The SIP service chain resulting from this composition is shown in Figure 4.24. After all three services were selected a command is given, which will result in indicating to SIP AS that there are no more services to be put on the chain and IMS will route the SIP invite to the destination in order to conclude the end-to-end session setup.

Please note that the application skeleton does not necessarily finish after instructing SIP AS to proceed with SIP message processing. However, no more SIP services can be selected, because the SIP service chain building was finished and dynamic additions and changes to an already established SIP service chain are not allowed according to IMS/SIP standards. The designer has to ensure compliant behavior. Nevertheless other types of services can be used. It would also be possible to capture further SIP messages within this application skeleton, thus within the same composition session.

4.5.3.1 Composition and control of SIP Service Chains

After the initial SIP service chain establishment of the end-to-end communication session, messages can be propagated over the SIP service chain both in forward and backward direction. Usually the services, which reside on the SIP service chain listen to the messages when they pass by. The services can autonomously decide if they ignore a message or if they become active. As an example, a service may become active by issuing a message itself or by altering the original message's content before forwarding it.

In addition, the designer may want to participate and take control over the signaling on the SIP session. This means, the composition core needs to be allocated as well in the SIP service chain to be able to intercept and evaluate signaling. The reception of a SIP message can be modeled as event reception in the application skeleton. Simply receiving an event about the message reception is not sufficient because it is very important to distinguish where on the end-to-end SIP service chain the message was caught. In order to control which events will be generated execution agents can be configured at the initial building of the SIP service chain to stay on the SIP service chain like services and to generate certain events, which can then be distinguished.

The skeleton in Figure 4.25 shows a complete use case with the SIP services Service1, Service2 and Service3 being put onto the SIP chain. Additionally two execution agent instances are allocated on the SIP service chain at different locations. These agents are configured to throw the events E1 or E2 if a message is received at their respective location. Two event handler bindings are specified in order to filter out SIP messages of type 486 BUSY and to bind the events to two different handlers. Although the message to be caught is the same, different event handlers are triggered depending on the location on the SIP chain where the message did appear.

Figure 4.26 shows the SIP service chain that corresponds to the example skeleton from Figure 4.25.

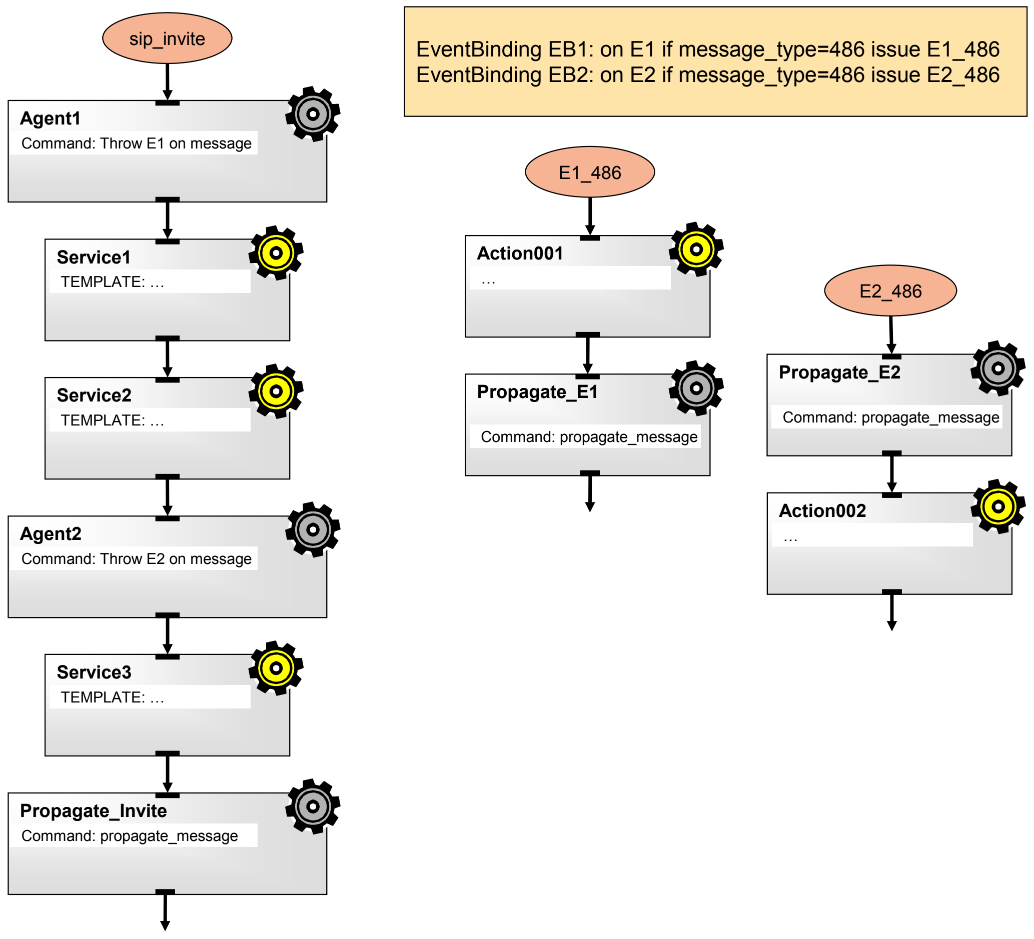
sip_invite
Agent1
Command: Throw E1 on message
Service1
TEMPLATE: ...
Service2
TEMPLATE: ...
Agent2
Command: Throw E2 on message
Service3
TEMPLATE: ...
Propagate_Invite
Command: propagate_message
EventBinding EB1: on E1 if message_type=486 issue E1_486
EventBinding EB2: on E2 if message_type=486 issue E2_486
E1_486
Action001
...
Propagate_E1
Command: propagate_message
E2_486
Propagate_E2
Command: propagate_message
Action002
...

FIGURE 4.25: Skeleton resulting in SIP service chains with agents for subsequent message handling

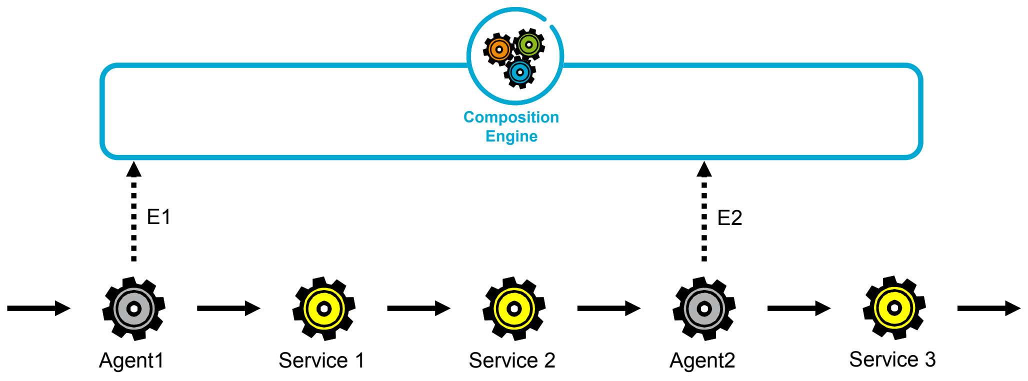

FIGURE 4.26: The resulting SIP service chain

The application skeleton can perform a couple of actions. The composition core can, in principle do all a service that resides on the chain is allowed to do. This includes for example evaluating and modifying the received message before it is propagated. The modifications will be fed back into the SIP chain at the location where the agent did break out with an event. It is also possible to initiate a partial release of the SIP service chain and start routing again towards a new destination with new services. The decision to do so, the control over the exact point where to break the chain and the composition of the new leg can be controlled in detail by means of skeleton elements.

Although the use of event generating agents was motivated by and demonstrated for SIP service chains, this mechanism is a generic one. It relies on the basic elements of SCaLE such as events, event handler bindings and actions to provide commands to the execution agents. These techniques can therefore also be used together with other service technologies. They are not bound to IMS/SIP.

4.5.4 Heterogeneous composition of SIP and Web Services

The examples shown previously have demonstrated how to use SCaLE in order to generate compositions for a couple of service technologies with different modes of service usage, taking into account their special characteristics. A heterogeneous composition would go one step ahead and allow composing services from various technologies within a single composition. SCaLE's main advantage is to permit this kind of development.

At its core query based service selection is focused on the provided function and agnostic to the technology which implements that desired function. The technological details will be taken into account only at service invocation rather than at service selection.

A simple heterogeneous example that combines web services and SIP services is shown in Figure 4.27. On SIP invite, an action is triggered that retrieves the user preference with respect to blacklist or whitelist services to be used for the SIP sessions. This service can for example be instantiated with a web service that retrieves this information from a user database. As a result, the preference of the user is stored in a variable. The action Black_or_whitelist evaluates the user preference and

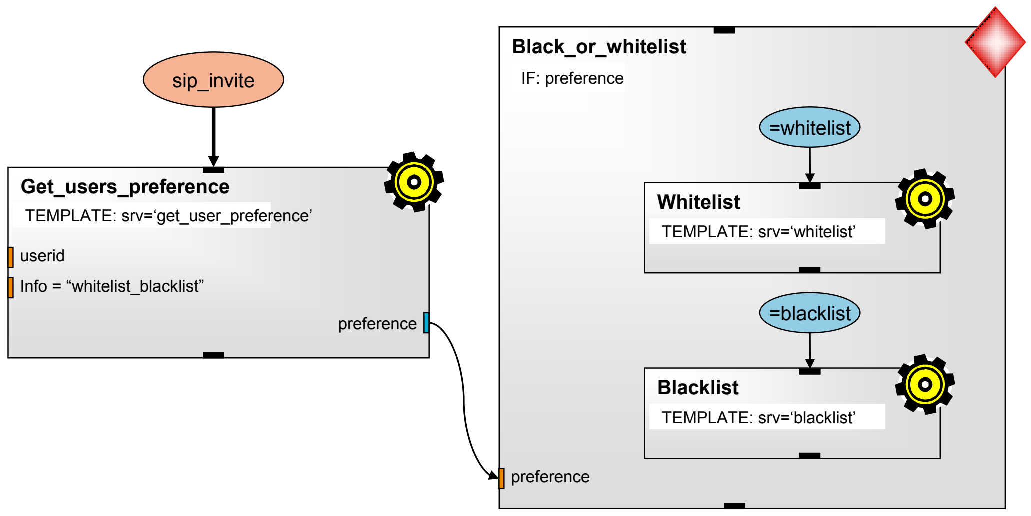

FIGURE 4.27: Blacklist or whitelist — a simple heterogeneous composition

executes the respectively selected service. As these are SIP services, execution means that the respective service is linked into the SIP chain.

4.6 SCaLE and WS-BPEL

This subsection attempts an early-stage use case specific comparison between SCaLE and WS-BPEL. More specifically we illustrate the same use case written in both languages as an early proof of how laconic SCaLE can be when expressing a fully-fledged use case from the telecommunications domain that combines services both from telecommunications but also from the ICT areas.

What is important to highlight here is that this comparison is done simply for the purposes of motivating in a hands-on manner, what are the benefits of writting in SCaLE as opposed to using WS-BPEL. However, this comparison is by definition limited to this particular example and cannot serve as proof for SCaLE's expressiveness. For a more complete comparison between SCaLE, WS-BPEL and BPMN 2.0 the reader should refer to 6.2 where the expressiveness of all three languages is assessed from a workflow pattern perspective.

The use case chosen for this example is a so-called "added-value" service, one can implement over the top in a telecommunications domain. This use case essentially allows for notifying a user about a missed phone call, when that user has enabled

Name	Type	Description
google_calendar_service	REST	Retrieves a user's calendar in XML
ReleaseControl	SIP	Part of SIP Execution Agent - Permits a SIP request to be released from the Application Server and reach it original destination
user_get_profile	WSDL	Internal WSDL service used to obtain a user's permission to access the external calendar
user_get_extid	REST	Internal REST service used for constructing the external calendar URI
find_available_slot	WSDL	Internal Service used for determining next available slot where both parties are available
SendSipResponse	SIP	Part of SIP Execution Agent - Constructs on the fly a SIP message that is disseminated later on in the IMS network
create_calendar_entry	REST	Used for submitting an XML snippet that represent a new entry in the Calendar

Table 4.1: Do-not-disturb use case service list

the do-not-disturb feature. The notification is done by means of creating entries in a Web based calendar. For the purposes of this example we have used Google Calendar. Moreover, this use case allows for rescheduling the phone call at a later point in time, one that is convenient to both parties based on their calendar information. More details on this use case can be found in [169].

The list of external services used by this example is shown in Table 4.1.

Since WS-BPEL does not support interaction with RESTful services, we have implemented WSDL based wrappers for all REST based services. Moreover, since WS-BPEL does not support Unified routing and therefore does not support the SIP Protocol, we have also created a WSDL based wrapper for our SIP Execution Agent (SIP EA).

We will first implement this use case in SCaLE. The implementation is shown in Figure 4.28

Execution order in SCaLE is by default parallel and asynchronous and it is implied by data dependencies. The composition framework will execute this application skeleton in the following order:

1. Wait for SIP INVITE

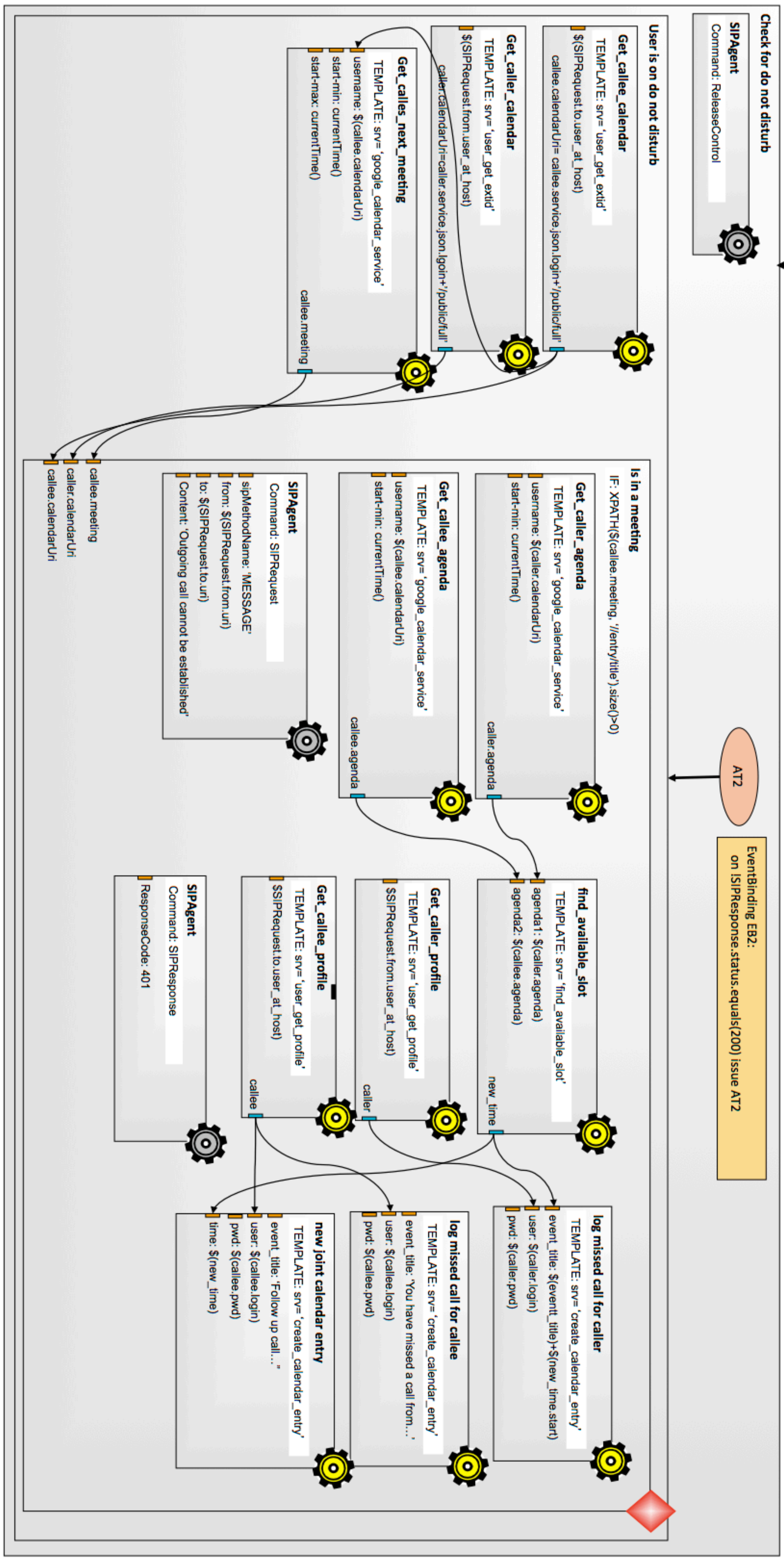
Check for do not disturb
SIPAgent
Command: ReleaseControl
AT1
EventBinding EB1:
on SIPRequest.method.equals('INVITE') issue AT1
User is on do not disturb
Get_callee_calendar
TEMPLATE: srv= 'user_get_extid'
$(SIPRequest.to.user_at_host)
callee.calendarUri= callee.service.json.login+'/public/full'
Get_caller_calendar
TEMPLATE: srv= 'user_get_extid'
$(SIPRequest.from.user_at_host)
caller.calendarUri=caller.service.json.login+'/public/full'
Get_callee_next_meeting
TEMPLATE: srv= 'google_calendar_service'
username: $(callee.calendarUri)
start-min: currentTime()
start-max: currentTime()
callee.meeting
Is in a meeting
IF: XPATH($(callee.meeting, '//entry/title').size()>0)
callee.meeting
caller.calendarUri
callee.calendarUri
AT2
EventBinding EB2:
on !SIPResponse.status.equals(200) issue AT2
Get_caller_agenda
TEMPLATE: srv= 'google_calendar_service'
username: $(caller.calendarUri)
start-min: currentTime()
caller.agenda
Get_callee_agenda
TEMPLATE: srv= 'google_calendar_service'
username: $(callee.calendarUri)
start-min: currentTime()
callee.agenda
SIPAgent
Command: SIPRequest
sipMethodName: 'MESSAGE'
from: $(SIPRequest.from.uri)
to: $(SIPRequest.to.uri)
Content: 'Outgoing call cannot be established'
find_available_slot
TEMPLATE: srv= 'find_available_slot'
agenda1: $(caller.agenda)
agenda2: $(callee.agenda)
new_time
Get_caller_profile
TEMPLATE: srv= 'user_get_profile'
$SIPRequest.from.user_at_host)
caller
Get_callee_profile
TEMPLATE: srv= 'user_get_profile'
$SIPRequest.to.user_at_host)
callee
SIPAgent
Command: SIPResponse
ResponseCode: 401
log missed call for caller
TEMPLATE: srv= 'create_calendar_entry'
event_title: $(event_title)+$(new_time.start)
user: $(caller.login)
pwd: $(caller.pwd)
log missed call for callee
TEMPLATE: srv= 'create_calendar_entry'
event_title: 'You have missed a call from…'
user: $(callee.login)
pwd: $(callee.pwd)
new joint calendar entry
TEMPLATE: srv= 'create_calendar_entry'
event_title: 'Follow up call…'
user: $(callee.login)
pwd: $(callee.pwd)
time: $(new_time)

FIGURE 4.28: Do-not-disturb use case in SCaLE

2. Wait for SIP Response
3. Get_callee_calendar
4. Get_callee_next_meeting, Get_caller_calendar
5. Get_caller_agenda, Get_callee_agenda, SIPAgent, SIPAgent
6. find_available_slot, Get_caller_profile, Get_callee_profile
7. log missed call for caller, log missed call for callee, new joint calendar entry

In SCaLE, this example uses two events/event bindings, 3 native commands towards the SIP Execution Agent and sends a total of 11 messages to 6 external services. To express this use case as an application skeleton SCaLE requires only 4 unique elements - 2 events/event bindings, 3 SIP EA commands, 1 if statement and 11 service action templates - therefore a total of 17 elements.

Next we will describe the use case in WS-BPEL using the notation found in Oracle Fusion Middleware [177], since WS-BPEL is deprived of standardized graphical notation. Since the default behavior for executing services in WS-BPEL is linear and synchronous (blocking-calls) we have intentionally tried to implement this use case in such a way to match the execution order that SCaLE produces by design. As a consequence, the same use case expressed in WS-BPEL is much lengthier due to the fact that in order to express an asynchronous call towards an external service, WS-BPEL requires at least three elements, one invocation, one wait element (in case block waiting is required) and one elmeent for receiving the response. In SCaLE asynchronous invocation is the default behavior. The same use case expressed in WS-BPEL in shown in Figures 4.29 and 4.30.

To express this use case as a workflow in WS-BPEL requires 5 unique elements - 10 <receive> elements, 14 <invoke> elements, 4 <wait> elements, 3 <if> elements and 2 <assign> elements - therefore a total of 33 elements.

The reason why WS-BPEL requires 3 additional invocations to external services as opposed to SCaLE is because in SCaLE the interaction with the SIP EA is a native command and therefore is not accounted as an external invocation. Another difference between SCaLE and WS-BPEL is the usage of the <assign> element. In SCaLE assigning values to variable can be done within the action service template

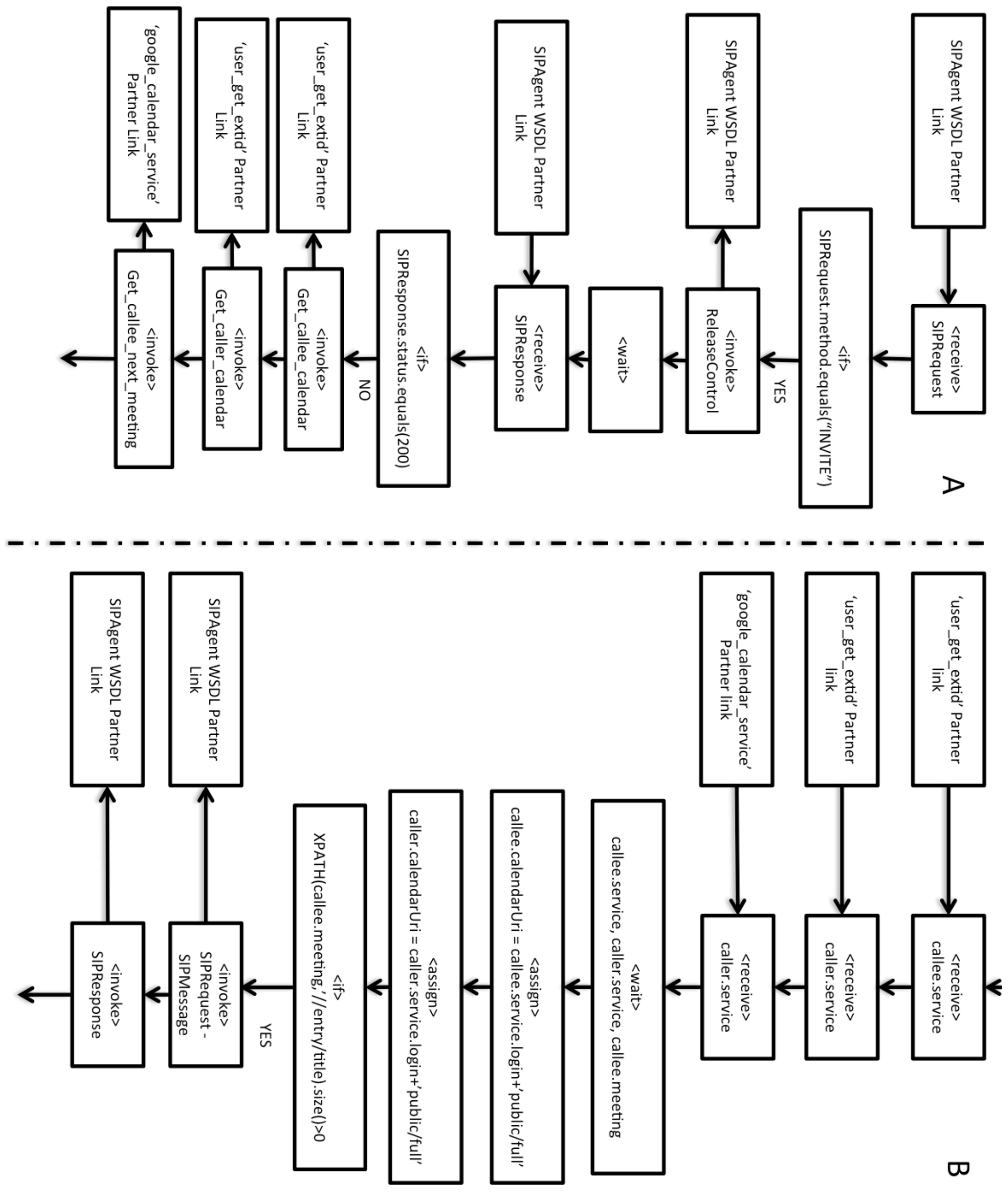
A
SIPAgent WSDL Partner Link
<receive> SIPRequest
<if> SIPRequest.method.equals("INVITE")
YES
SIPAgent WSDL Partner Link
<invoke> ReleaseControl
<wait>
SIPAgent WSDL Partner Link
<receive> SIPResponse
<if> SIPResponse.status.equals(200)
NO
'user_get_extid' Partner Link
<invoke> Get_callee_calendar
'user_get_extid' Partner Link
<invoke> Get_caller_calendar
'google_calendar_service' Partner Link
<invoke> Get_callee_next_meeting
B
'user_get_extid' Partner link
<receive> callee.service
'user_get_extid' Partner link
<receive> caller.service
'google_calendar_service' Partner link
<receive> caller.service
<wait> callee.service, caller.service, callee.meeting
<assign> callee.calendarUri = callee.service.login+'public/full'
<assign> caller.calendarUri = caller.service.login+'public/full'
<if> XPATH(callee.meeting,'//entry/title).size()>0
YES
SIPAgent WSDL Partner Link
<invoke> SIPRequest - SIPMessage
SIPAgent WSDL Partner Link
<invoke> SIPResponse

Figure 4.29: Do-not-disturb use case in BPEL, A-B

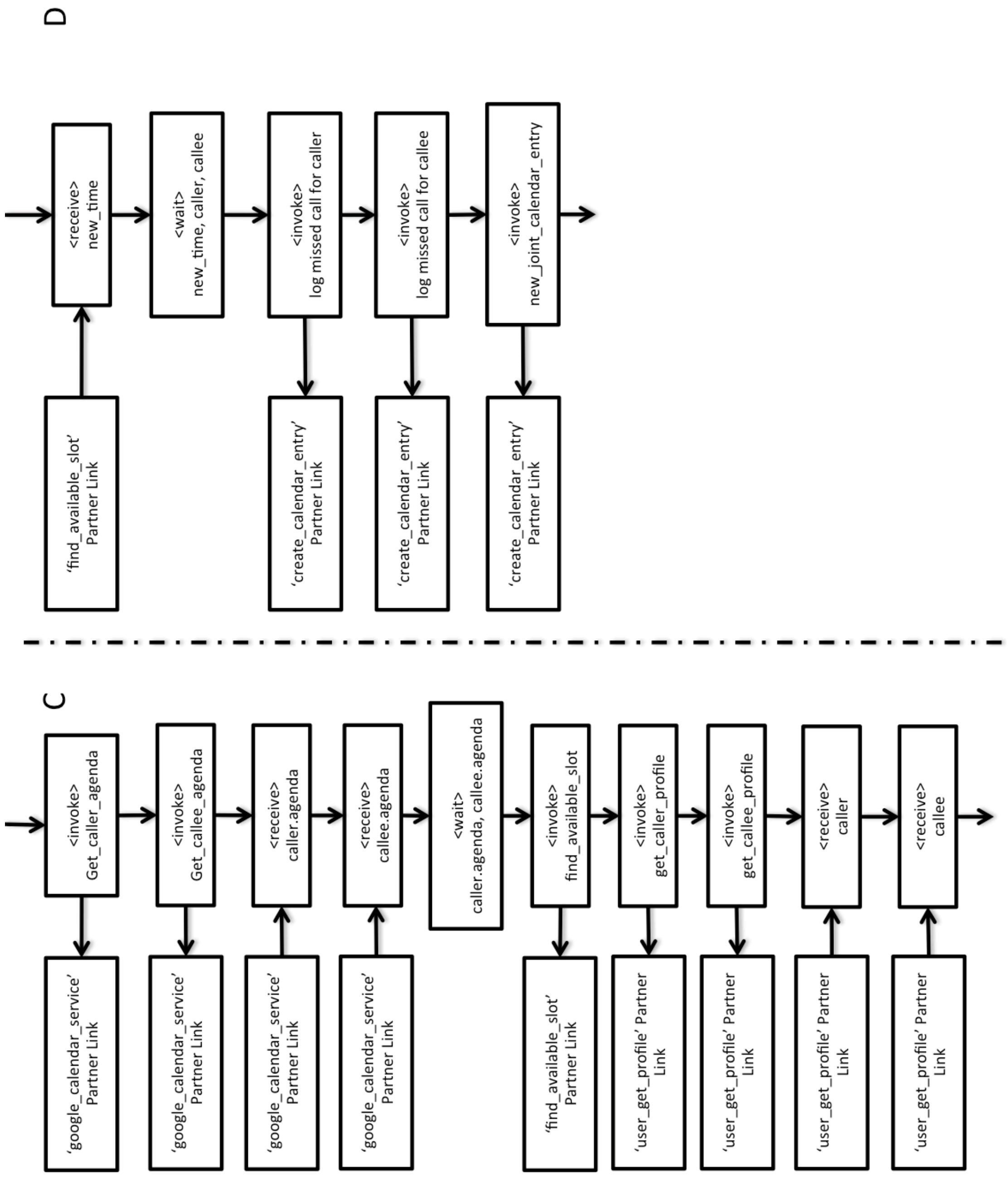
D
<receive> new_time
'find_available_slot' Partner Link
<wait> new_time, caller, callee
<invoke> log missed call for caller
'create_calendar_entry' Partner Link
<invoke> log missed call for callee
'create_calendar_entry' Partner Link
<invoke> new_joint_calendar_entry
'create_calendar_entry' Partner Link
C
<invoke> Get_caller_agenda
'google_calendar_service' Partner Link
<invoke> Get_callee_agenda
'google_calendar_service' Partner Link
<receive> caller.agenda
'google_calendar_service' Partner Link
<receive> callee.agenda
'google_calendar_service' Partner Link
<wait> caller.agenda, callee.agenda
<invoke> find_available_slot
'find_available_slot' Partner Link
<invoke> get_caller_profile
'user_get_profile' Partner Link
<invoke> get_callee_profile
'user_get_profile' Partner Link
<receive> caller
'user_get_profile' Partner Link
<receive> callee
'user_get_profile' Partner Link

FIGURE 4.30: Do-not-disturb use case in BPEL, C-D

element, directly on the input or effect variables, while WS-BPEL requires the usage of an additional element. Finally yet importantly, SCaLE can combine an if statement with an event binding, therefore there is no need for an additional element in order to evaluate a condition.

4.7 Summary

In the beginning of this chapter we have defined our design objectives for the proposed composition framework and the language we coined in order to enable a designer to compose in a heterogeneous service environment.

We started with the design of the proposed composition framework and its constituent components. These components are the composition core that is responsible for interpreting SCaLE, the application skeleton repository which is a repository of application skeletons, developed by the various designers using this system, the services repository which contains a list of service description for external services, a list of pre-existing functionality that can be re-used through queries by an application skeleton, a shared state where runtime information is stored, a set of execution agents (HTTP, SIP, WebSockets, IN, ESB)) and last but not least an Integrated Development Environment which allows for graphical development of compositions.

We then continued our description by detailing the constructs of SCaLE, which is the language that allows for the definition of compositions as application skeletons. There are three key characteristics in the runtime semantics of the proposed language that differentiate it from the state of the art in workflow or workflow like languages.

1. By default, the execution of individual actions, defined in the language is governed by data dependencies unless explicit ordering is enforced
2. Instead of strictly typed function calls, queries are used in order to re-use pre-existing functionality which is implemented externally
3. Once a composition is triggered a new session is created that collects all runtime information. Sessions can be re-used for the purposes of data exchange.
4. An optional type system is employed that allows the designer to dynamically moved from a dynamic to a static type system.

We conclude this chapter by providing a set of examples that show how the proposed framework and language can be used to describe services to support either stand alone external service interaction patterns, or intertwined permutations of those. Three main interaction patterns are identified: synchronous request-response (4.5.1), asynchronous message (4.5.2) exchange and service chaining (4.5.3). We provided an example for each interaction pattern and finally an example that combines these patterns in one (4.5.4).

Table 4.2 is a reiteration of Table 3.1 from Chapter 3. In this re-iteration we have added a column that places our proposed approach in comparison to the approaches for service composition introduced in Chapter 2.

		Static approaches	AI planning	Domain Specific Languages	SCaLE
Design-time	Design to runtime relationship	Direct (uses common language constructs to define composition)	Implicit (planning problem is transformed to a composition)	Indirect (domain specific constructs are related to general purpose instructions)	Direct (uses common language constructs to define the composition)
	Formal Semantics	Most cases	All cases	Most cases	Yes
	Runtime objects	New objects can be created at runtime	All objects are available in the initial state – new objects are not tolerated at runtime	New objects can be created at runtime	New objects can be created at runtime
	Service availability	Lack of suitable services in the service repository would break the process at runtime	Lack of suitable services in the service repository breaks the planning process at design-time	Lack of suitable services in the service repository would break the process at runtime	Depends on the type system
Runtime	Service Invocation	Limited to a particular technology			Agnostic via execution agents
	Type System	Static type system			Optional type system
	Unified routing	Not supported			Supported by design
	Concurrency	Explicit definition			Implicitly defined
	Shared State	Not supported			Supported

Table 4.2: Assessment of approaches to service composition including SCaLE

Chapter 5

Implementation

In this chapter we describe the implementation details of the different components that constitute the proposed framework. In a way, the implementation description of SCaLE can be viewed as an experience report on the usage of the proposed composition framework and the implementation description of the composition framework can be seen as an evaluation of SCaLE usage for practical applications. The implementation was done in order to demonstrate the viability of the proposed approach. This chapter begins by providing an overall description of the implementation of each component of the proposed service composition framework. Then we proceed to describe in detail the implementation of each component and most importantly, the implementation of the execution agents.

Moreover, we utilize the explanatory powers of five practical measures of architectural complexity and with the aid of static analysis tools we attempt to evaluate the quality of the source code base that was developed. The following software metrics [129] are used in order to quantify the size and complexity of this work. More specifically, we measure software lines of code (SLOC), cyclomatic complexity [65], first-order density [103], propagation cost [103] and finally core-size [156].

5.1 Overall description

Our implementation goal is two-fold. On the one hand-side we are interested towards a highly performant modularly implemented system and on the other hand side we are interested in producing a source code base that is easily understood and maintained by developers. These particular goals are met by Java and JavaEE, taking into consideration the maturity of this particular language and supporting tool but also the popularity of the language. According to the TIOBE programming Community Index [8], Java ranked as the 1st most popular language in 2008, a position that was later on taken by C, making Java the 2nd most popular language.

In addition, usage of JavaEE-based SIP application servers has been a common way to develop applications for the telecommunication domain as early as 2003 with the introduction of the JSR116 [94] specification, which was later on updated to JSR289 [175] in 2008. The benefits of using a SIP application server (also known as a SIP container) are:

- Usage of a development process that follows a clearly defined and standardized approach,
- Usage of the Java programming language (well established programming language with garbage collection and easy to use threading model, controlled by the Java Virtual Machine),
- Re-usability of SIP listeners, controlled by the application server itself.

Essentially, such a setup allows a developer to focus solely on the business logic of a SIP application and on how to handle SIP signaling, instead of worrying about the SIP state machine and how the SIP traffic can reach this application to begin with. As the usage of application servers became more widespread, so did the features provided by them. More specifically, application servers were enhanced to allow for converged SIP applications that would allow mixing signaling from HTTP with SIP and vice-versa.

5.2 Composition framework

This section reveals the implementation details of the proposed composition framework. More specifically, the following components and interaction between them are analyzed in detail:

- Composition core
- Application skeleton repository
- Services repository
- Shared state
- Execution Agents
  - HTTP
  - SIP

As mentioned in Chapter 4, additional components are included in the proposed composition framework such as execution agents for WebSockets (Web Sockets EA), Intelligent Networks (IN EA), a JBI Execution Agent for interacting with Enterprise Services Buses (ESBs) and finally yet importantly an Integrated Development Environment (IDE). Even though these entities have been implemented and are part of the proposed composition framework, their design and implementation descriptions is omitted from this thesis. More specifically we focus only on describing the HTTP and SIP EAs since these two put together, formulate a design complexity space between a simplistic execution agent (HTTP EA) and a complex one (SIP EA). Therefore all other execution agents, from an implementation complexity point of view, would fall somewhere in between and moreover would follow a rather streamlined implementation process in order to communicate with the proposed composition framework. The design and implementation of an IDE for the proposed language is omitted since we are mostly interested in examining the computational properties of the proposed framework and language and not the aspects of the tool that utilizes them.

5.2.1 Composition Core

Our main design goal is to design a composition core that adheres to an asynchronous, event-driven, non-blocking paradigm.

While JavaEE is a rather mature technology with rich APIs for developing enterprise-grade systems, we have experienced certain limitations when it comes to the implementation of highly scalable, asynchronous systems. Specifically, before the introduction of the "@Asynchronous" annotation in EJB 3.1 [139], all EJB invocations were handled as synchronous, blocking calls, which manifested themselves in a thread-per-request style of requests processing. The maximum number of available threads (i.e. maximum number of threads in the Java Virtual Machine (JVM) [100] or application server) is a known limiting factor with regards to scalability. In addition, this limitation may lead to starvation under very high load or if there are many long running requests. Alternatively, Message Driven Beans (MDBs) [146] in combination with JMS [80] can better support requirements for asynchrony, but tend to run into similar scalability issues, especially under heavy load as experienced in typical telecommunication applications.

In the context of our work, applying a thread-per-request model would entail that dedicated threads handle requests; at least one thread would be responsible for interpreting an application skeleton until its completion, after which the thread would be released and made available to other requests. Moreover, invocations of external services would block the thread until a result was available. In such a setup, the amount of composite applications that can be run in parallel is directly proportional to the amount of available threads. Therefore, the maximum number of threads supported by the runtime environment would limit the number of different application skeletons that can be interpreted simultaneously. In the case of JVM, this limit is typically around 15000 or 20000 threads.

This observation motivated us to migrate from a thread-per-request model towards a mapping of requests to a set of fine-grained executable tasks that can be scheduled on any number of threads. In addition, it led us to make use of non-blocking I/O approach for any invocations of services (e.g. SOAP/REST/JBI services), which avoids blocking the caller thread for the duration of the I/O operation.

Techniques for writing software that operates in non-blocking asynchronous fashion have been well established in various fields of computer science such as development of operating systems [159]. In addition, similar techniques have already been implemented in research implementations of BPEL engines such as BPEL-Mora [73] and even into HTTP listener infrastructure such as Grizzly/Comet [105].

The composition core interprets application skeletons by dividing them into small tasks. Several pools of worker threads that are not tied to a specific application skeleton process these tasks. Whenever an application skeleton has to wait for the execution of an I/O operation, the corresponding task is put on a waiting list and will be retrieved by a worker thread along with its context information when the result of the external invocation is available. In addition, some improvements were made in the area of the support for external protocols such as Grizzly/Comet in HTTP to allow for deferred response notification. These improvements permit the composition framework to process a very large number of HTTP or SIP -triggered application skeleton.

Practically speaking for this to be achieved at the source code level, every large and synchronous method needs to broken down into smaller pieces. The choice of finding at which point a method should be broken into smaller sub-methods was made by looking into those segments of code where blocking waiting could occur. Each resulting sub-method is submitted as a task to the thread pool and executes sequentially, but when it reaches the break, that is the place where the original method would need to wait for the results of a long-lasting operation or an external service invocation, it would register a continuation callback.

The continuation callback creates and submits a task to execute exactly the subsequent sub-method corresponding to the code of the original method after the break. In addition, it preserves the current state of the application skeleton, starts the long lasting operation and suspends the execution of current application skeleton by releasing the thread it currently occupies. In this way, the thread is made available for executing other application skeletons. Once the long-lasting operation is over, the registered callback would be triggered, which allows for the application skeleton interpretation to resume by submitting the next task.

The execution of composite applications is now represented as a set of small tasks executed by thread pools. Due to this decoupling from threads, the maximum number of threads or the number of threads in the thread pools is not important any more. Any number of simultaneous application skeletons can be executed even by a thread-pool with a very small number of threads. This comes at the expense of consuming more memory by tasks and the suspended state of application skeletons. Therefore, the amount of available memory for storing this information may become a limiting factor. But even with current commodity hardware such as the one used for our testing purposes, this implementation allows for hundreds of thousands or even millions of application skeletons to be handled by one machine, in contrast to 20000 composite applications that has been the limitation of a thread-per-composition approach.

This style of conversion described here is a well-known transformation called Continuation Passing Style (CPS) [144]. It is often used in functional programming. Functions written in this style never return; instead they always pass what should be done next (therefore it is called "continuation") as a parameter to the functions they invoke. Due to this design, any computation can be easily suspended or resumed at any moment, because continuations contain everything that is required to proceed with the computation.

Usually, the compiler does CPS transformations automatically. But in the case of Java-based systems, since Java/JVM does not support continuations, this transformation was done manually and it has been a very difficult task to keep track of this conversion. In particular, one should be very careful with providing correct callbacks, i.e. continuations, and preserving/restoring the state when suspending and resuming the execution of current composite application.

We had to manually convert all uses of local variables (allocated on the call stack) into uses of variables allocated on the heap and thus surviving thread releases, when call stack is un-winded by the JVM and information stored on it is lost. Essentially, this part of the transformation was similar to a combination of taking a closure of current context combined with lambda lifting. Lambda lifting [88] is often used by automated CPS transformations, where variables from different nested scopes are promoted into the top-most scope, which would be the heap in this case.

The resulting composition core can be deployed in two ways: inside a JavaEE container (usually the application server SailFin) or standalone. In the latter case, we provide our own northbound execution agents instead of relying on the infrastructure provided by the container. This kind deployment reduces the overhead that is usually added by the application server and allows a more direct management of threads and resources.

5.2.2 Application skeleton repository

The application skeleton repository is a container for application skeletons (or skeletons). An application skeleton can be understood as a blueprint or pattern for a composite service. The composition core typically uses deployed application skeletons, available in the application skeleton repository, as the basis for creating appropriate composite services. The creation of a composite service based on an application skeleton is referred to skeleton execution.

As previously described in 4.4.1 an application skeleton defines:

- The set of participating services
- Structure, i.e. how individual services should connected to composite service,
- Control flow, i.e. in which order individual services should be executed

The designers create application skeletons manually. An application skeleton is static and cannot be altered by the composition core or by services. Concepts found in genetic programming [93] are intentionally absent from our proposal for the purposes of ensuring deterministic behavior. However, it is allowed to make execution of skeleton elements depending on runtime information. So that skeleton elements will be taken into account during skeleton execution only under certain prerequisite conditions. Furthermore, skeleton elements can be grouped into branches whose execution can be also dependent on runtime information. Examples for possible runtime information are user preferences, skeleton execution status, network status, allocated resources, etc.

Application skeletons can contain descriptions of alternative services that can be used as replacements for cases where the original service cannot be successfully executed. Alternative services will be executed one-by-one until one of them is successfully executed.

These mechanisms make application skeletons more adaptable to runtime context so that once designed skeleton can be executed repeatedly for a long time period and in different contexts. A consequence of alternative services and elements depended on runtime information is that different composite services can be derived from one

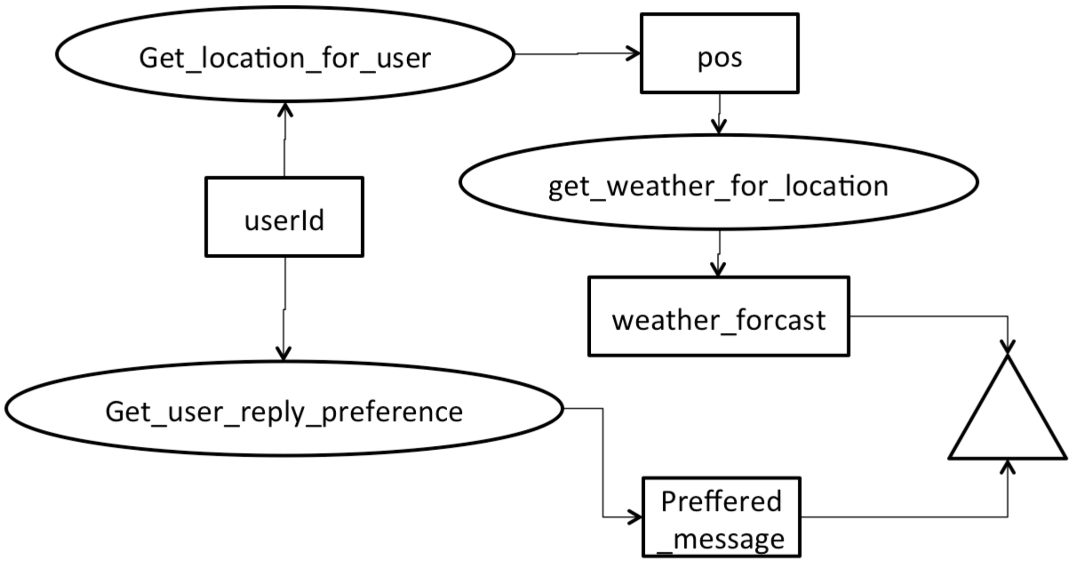

FIGURE 5.1: Low-level representation of weather information application skeleton

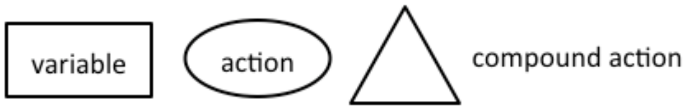

FIGURE 5.2: Legend for low-level representation symbolism

skeleton. An application skeleton is represented as a directed acyclic graph, as it schematically shown in Figure 5.1.

In practice all the nodes are represented as circles in the underlying system that we use to store this information. However, in Figure 5.1 we use rectangles to represent variables, circles to represent actions and triangles to represent compound actions for the purposes of readability.

An edge originating from a variable and ending in a variable signifies that the action depends on that variable while an edge originating from an action and resulting into a variable means that the action produces that variable.

The representation shown in Figure 5.1 is the actual information that is being stored in the graph database and it has been produced as an example from Figure 5.4. As such, the linguistic graphical symbolism presented in the previous section can be viewed as syntactic sugar. This choice was made intentionally in order to simplify the process of designing the application skeleton.

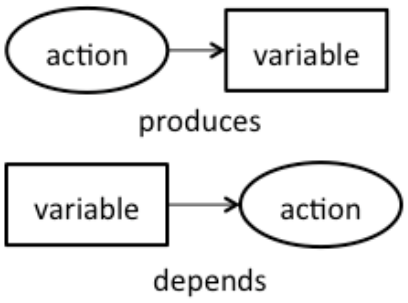

FIGURE 5.3: Edge-node, node-edge directed relathionship

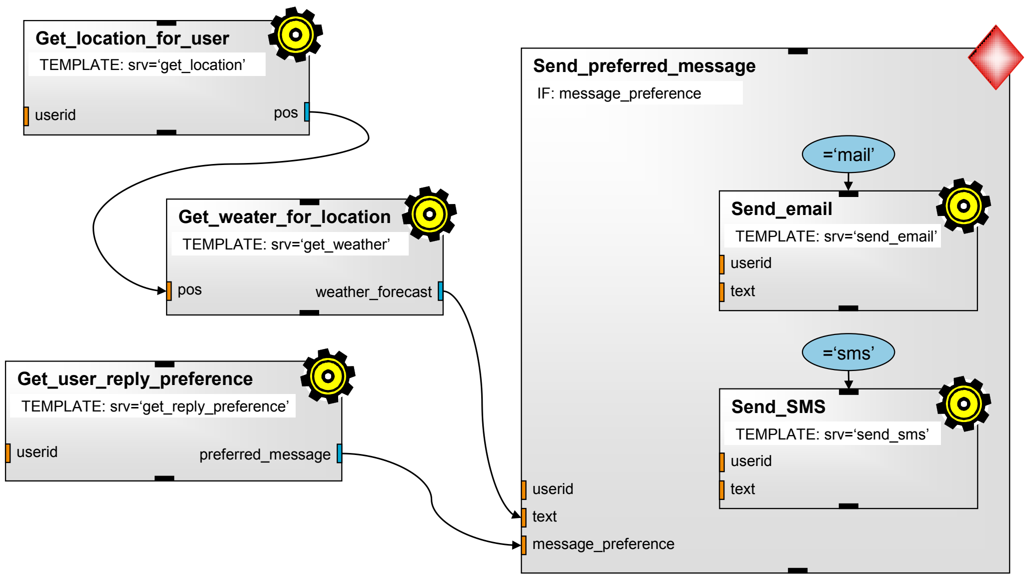

FIGURE 5.4: Weather information application skeleton

Figure 5.5 explains how send_preferred_message is modeled.

5.2.2.1 Graph databases

Graph databases are based on graph theory and as such, they represent information by using nodes, edges and properties. By definition, a graph database is any storage system that provides index-free adjacency. This means that every element contains a direct pointer to its adjacent element and no index lookups are necessary.

We chose to use a graph database instead of a relational database for the purposes of the application skeleton repository since a graph database, offers faster access to associative data sets and maps more closely to the nature of the application skeleton. As such it scales naturally to larges sets of data as it is deprived of join

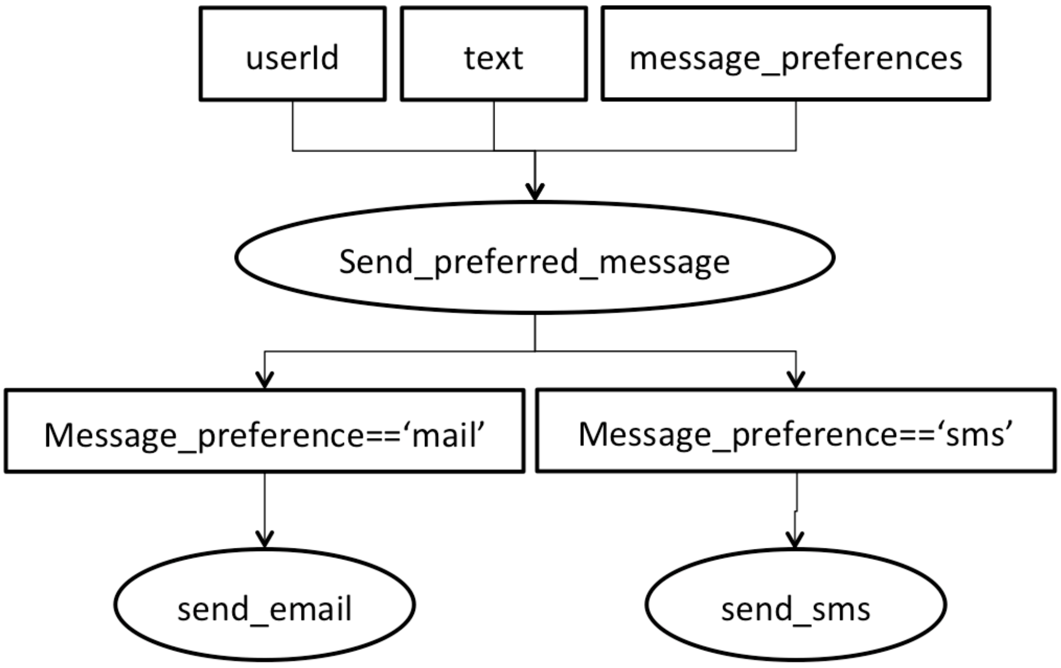

Figure 5.5: Low-level graph representation of weather information application skeleton

operations [137]. In addition, it does not depend to a schema and this makes it more flexible with changing data. In addition, the graph database is a powerful tool for computing graph like queries such as computing the shortest path [57].

Several implementations of graph databases exist, Neo4J [118] and OrientDB [123] being the most prominent ones. Angles et al. [6] provide a comprehensive review of graph databases. In our case, since we do not want to adhere to a particular graph database, we chose to use blueprints by Tinkerpop [138] that allows for agnostic access to any graph database that supports the blueprints generic graph API. For the purposes of the prototype development we chose to use Neo4J along with blueprints due to the maturity of that particular graph database implementation and the large community around it.

5.2.2.2 Service template action

A service template action is a placeholder for one specific service, more precisely for a reference to one specific service. This service will be selected at runtime from the list of compatible services that will be created based on the information provided by the service template. A list of compatible services can be defined in two ways:

- An enumeration of compatible services or
- A description of the properties of compatible services.

An enumeration consists of direct references to specific services so that these services can be taken over directly into a list of compatible services

A description of properties has a query like form that resembles a regular expression. All services that fulfill the query of the service template action are included in the list of compatible services. An enumeration can be modeled as follows:

```
(srv=id_of_first_service_in_enumeration) | (srv=id_of_second....)
```

This allows combining enumerations and queries in one service template. In addition to queries or enumerations a service template can contain service attributes (subsection 5.2.4.1), which will be merged with service attributes of the service. As service constraints (global and local) are also defined using service attributes it is possible to define additional service constraints in the service template. This way the properties of a service can be changed through its participation in a composite service. However, additional attributes of a service inherited from a skeleton element apply only in the scope of this composite service and skeleton, i.e. other composite services and skeletons do not see these additional attributes. As an example, an additional service attribute, which describes the position of a service in a composite service (see example 4), describes the position of service only in a specific composite service. What service attributes can be added to a service template is defined by service class service_template.

Independent from service templates and from service constraints and attributes the selected service has no impact on previous, i.e. already selected and executed, services. Constraints only impact the selection of a service for current and for the next service templates in scope of current composite service. This makes global constraints in service templates applicable only to the next skeleton elements, which are not yet executed.

A particular case of service template is a service template without any constraints and with empty enumeration. Such service templates cannot be used for describing any particular service but they can be useful for defining additional properties and limitations for a composite service, e.g. global constraints can be defined in such service templates.

The following examples (1-6) illustrate the use of service templates in Polish prefix notation [60].

1. enumeration

```
[serviceA, serviceB, serviceC, serviceD]
```

   list of compatible services: serviceA, serviceB, serviceC, serviceD

2. with a constraint

```
[ (&(|(service_provider=providerA)(service_provider=providerB))(
    function=VoIP))]
```

   list of compatible services: all services from providerA and providerB with VoIP functionality.

3. with enumeration and a constraint:

```
[|(srv=serviceA)(srv=service)(function=VoIP))
```

   list of compatible services: serviceA and serviceB and all services with VoIP functionality.

4. with additional attributes

```
(|(srv=serviceA)(srv=service)(function=VoIP))
    service_position_in_composition:3
```

   list of compatible services: the additional service attribute describes the position of the service in the composite service. The service that will participate in the composite service gets this additional attribute from a service template

5. with additional constraints

```
(|(srv=serviceA)(srv=service)(function=VoIP)) global_constraint:
    (provider=providerA) constraint: (function=white_list)
```

   list of compatible services: The additional constraint permits the combination of the selected service with the services with function white_list and enforces that all services in the composite service belong to providerA.

6. Skeleton objects

```
[global\_constraint: (provider=providerA)]
```

list of compatible services: All following services must belongs to providerA

In most cases the results of the evaluation of the service templates is the execution of at least one service so that a service element has impact outside composition core. In contrast, the element described in the next chapter only has impact inside the composition core.

5.2.2.3 Parameter passing in service action templates

This subsection details the inner-workings of the process of passing input parameters to external services when using service action templates. Essentially, this process attempts to solve the problem of associating input parameters to selected services to be invoked.

Assume that the result of dynamic service selection, implemented by means of a service template action has returned three different results for a certain query.

- Result 1: foo(XMLDocument1 doc1)
- Result 2: foo(JSONDocument1 doc2)
- Result 3: foo(CustomPayload doc3)]

This implies that three different services are applicable to a particular query. Unlike normal programming languages, where function invocation needs to deal with placing each input to a particular argument placeholder, in our case, since we deal with external services we need to take care of placing the designer's input parameter to the correct placeholders within different kinds of payloads, such XML, JavaScript Object Notation (JSON) and custom user made payloads. Placing placeholders in each document, in the particular points where designer's input needs to be placed, does this.

Consider listing 5.1 as part of XMLDocument1:

```
<note>
<to>${to}</to>
<from>${from}</from>
<heading>${heading}</heading>
<body>${body}</body>
</note>
```

Listing 5.1: Annotated XML Snippet

In this snippet you will notice four distinct placeholders marked with ${ and }. By placing these placeholders, the composition core knows at runtime where to put the contents for runtime variables to, from, heading and body.

The same principle applies to JSONDocument1 in listing 5.2

```
{
"employees": [
{ "firstName":"${employeeName1}" , "lastName":"${employeeLastName1}"
    },
{ "firstName":"${employeeName2}" , "lastName":"":"${employeeLastName1
    }" },
{ "firstName":"${employeeName3}" , "lastName":"":"${employeeLastName1
    }" }
]
}
```

Listing 5.2: Annotated JSON Snippet

Placeholder information is set in the service description for reach external service.

5.2.2.4 Conditional elements

A skeleton condition is an expression that will be evaluated at runtime using runtime information. Depending on the evaluation results a skeleton branch will be selected. Thus providing conditional elements with better control over skeleton execution and allowing the creation of composite services depending on runtime variables.

The composition core provides a set of runtime variables that can be used to evaluate conditional statements. Such information includes the original request, the user profile, the participating devices, the current status of skeleton execution and the

Operands / Result Type	Logical	Numerical	Textual
Logical	OR, AND, NOT, XOR	=, <, >	=, contain
Numerical		+, -, *	length, position
Textual			upper_case, sub_string

TABLE 5.1: Operations applicable to runtime variables

composite service. Runtime variables can have logical, numerical or textual values. The usage of runtime variables is described in detail in subsection 5.2.3.1. The operations in Table 5.1 can be used with runtime variables depending on their values.

In addition to these operations the usage of parentheses in conditional expressions is allowed. The invocation of services from conditional expressions is not allowed.

The value of runtime variables cannot be changed directly by skeleton elements. However, the execution of skeleton elements may result in the modification of information regarding the skeleton execution, e.g. the currently processed element, execution results, etc. Skeleton elements may therefore have indirect impact on runtime variables. Furthermore, individual services can create new and change existing runtime variables. This way services can influence indirectly the creation of a composite service.

For each possible result of a conditional expression a skeleton branch must be defined. The definition of a default branch is allowed. If no branch corresponds to the result of the conditional expression, the skeleton execution is terminated.

The following (1-3) examples illustrate alternatives for conditional expressions:

1. A logical condition with the runtime variable request type, which is TRUE if the original request was a SIP request.

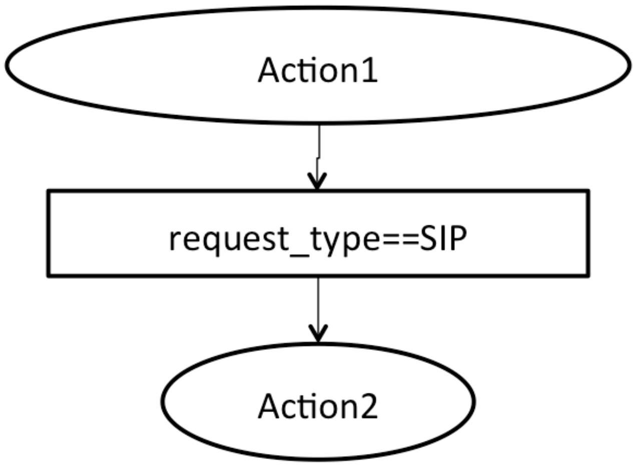

Figure 5.6: Simple logical condition

2. A more complex condition that will not allow a user with a prepaid tariff to have more than 3 parallel composite services.

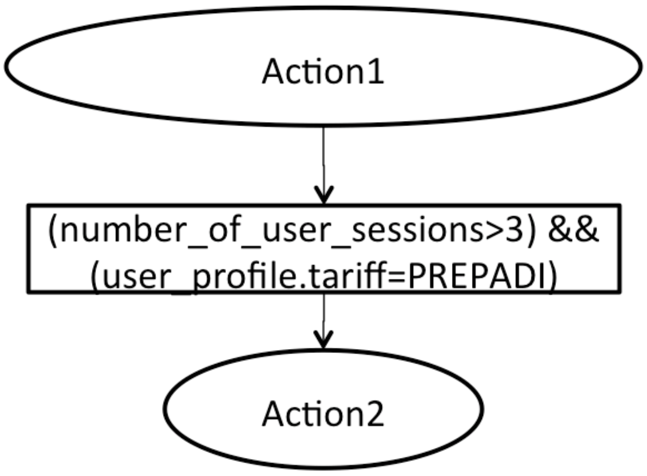

Figure 5.7: Complex logical condition

3. A textual condition with a default case that selects a branch depending on user tariff.

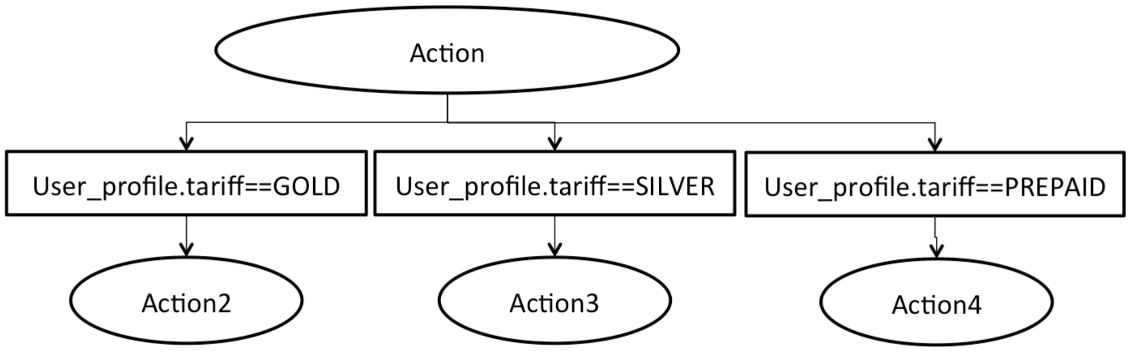
Action
User_profile.tariff==GOLD
User_profile.tariff==SILVER
User_profile.tariff==PREPAID
Action2
Action3
Action4

FIGURE 5.8: Textual condition

5.2.3 Shared State

Within the composition core, the so-called shared state of a composition session plays the central role for the coordination of an application skeleton and constituent service execution. All acquired data from the execution agents involved in a composition session are stored in the shared state, as well as intrinsic data of the execution environment. For example the parameters of SIP messages are stored within the shared state. Thus, information such as user addresses is known to the composition core and can be provided as input to services that demand user profile settings or user locations. Another use of this data might be the manipulation of the SIP message itself that is sent back to IMS. For example, the target address could be changed in order to redirect the session. Any change in the shared state produces a high-level event related to it, e.g. change of the variable's value generates an event "StateChanged(variable name, old value, new value)". These events are means of communication between components using different technologies. The Shared State Manager employs a subscribe/notify model. Every participating component of composition can subscribe to any set of possible events and will be notified about such events. Therefore, the Shared State Manager manages both variables and their values, e.g. key-value pairs and could include event queues or something similar. It is possible to define new variables and their values, update the values of variables and read the values of variables.

For this reason we have chosen to go with a NoSQL key value store approach since our main motive is to reap the benefits in terms of latency and throughput. In particular we chose to use Apache Cassandra [95], which is actually a hybrid between a key value store and a row-oriented database.

5.2.3.1 Session Information

The information available in the shared state manager is grouped per session and it is referred to as Session Information. This information is created at runtime and consists of a set of variables that keep track of the current state of the application skeleton. These variables are referred to as runtime variables. Session information is by default private to the application skeleton that handles the request that resulted into the creation of the session to begin with. However, there is a special session called global whose information can be accessed by any other session. An identifier of a variable in a global session always stars with the prefix ”global”, i.e. global.user_session_count. Table 5.2 provides an example of session information variables that are created by default for every application skeleton instantiation. The purpose of such variables is to keep track of a skeleton's execution state.

Variable id	Variable Type	Description	Example
skeleton	String	ID of initial Skeleton	voipSkeleton
skeleton.last_element	String	ID of last evaluated skeleton element	busy_check
skeleton.last_element.skeleton_id	String	ID of skeleton of last evaluated skeleton element	voipSkeleton
skeleton.last_element.type	String	Type of last evaluated skeleton element	SERVICE TEMPLATE
skeleton.last_element.result	String	Result of last evaluation (uri of service or condition result)	sip@server.com
skeleton.next_element	String	ID of the next to be evaluated element of skeleton	checkUserProfile
skeleton.next_element.type	String	Type of the next to be evaluated element of skeleton	CONDITION
skeleton.executed_elements_count	int	Count of executed skeleton elements	3
skeleton.executed_element1	String	Id of first executed skeleton element	skeleton1
skeleton.executed_element1.skeleton_id	String	ID of Skeleton of the first element	voipSkeleton
skeleton.executed_element1.type	String	Type of the first executed element	START
skeleton.executed_element1.result	String	Result of evaluation of the first executed element	
skeleton.executed_element2	String	Second executed skeleton element	busy_check_wl
skeleton.executed_element2.skeletonId	String	ID of Skeleton of the second element	voipSkeleton
skeleton.executed_element2.type	String	Type of the second executed element	SERVICE TEMPLATE
skeleton.executed_element2.result	String	Result of evaluation of the second executed element	busy_check_wl@server1.com
user_profile	String	internal ID of user profile	bob_profile
user_profile.user_type	String	Type of user e.g. normal, gold, etc	normal
user_profile.preferred_audio_service	String	Preferred audio service e.g. voip, ptt, etc	ptt

TABLE 5.2: Default runtime variables

Moreover, external services and applications can define additional runtime variables and provide on this way their runtime information to the composition core and to composite services. Further it is conceivable that services (e.g. Voice Over IP (VoIP) server) can provide information about its status using an interface to these objects (e.g. the VoIP object) and the skeleton can include conditions, in which this status will be evaluated. An external runtime variable is independent from the composite service and is identical in all runtime environments. The external runtime objects allow better integration of external services into composition process. However, the skeleton designer must know these external services and their objects. Runtime variables are referenced from a skeleton and service descriptions using this form:

```
${object_identifier.variable_identifier}
```

Example:

```
${user_profile.tariff}
${skeleton.last_element.result}
```

5.2.4 Services repository

An important aspect in the overall process of service composition is the availability of a repository of service descriptions that describe the characteristics of external services that can be re-used. More specifically a service description contains information that is useful for an execution agent, with regards to how to access an external service and information regarding the properties of an external service that is useful for the designer.

The composition core has access to information on the requirements of all services involved as well as the functionality they are in a position to provide to other services. Since the internal functionally of a service is hidden from the composition core and the information that can be extracted from established service descriptions, e.g. SDP [79] or WSDL, is not enough for the overall composition on a functional level, additional information about service properties is needed. This information is modeled using service attributes and service constraints. Service attributes describe what a service provides and service constraints describe what a service requires. These two modeling mechanisms are discussed in the 5.2.4.1 and 5.2.4.2 respectively.

srv	ServiceA
URI	sip://server.com/voip
service_provider	ProviderA
function	audio_transmition
audio_codec	GSM
audio_codec	G.771
description	VoIP service

Table 5.3: Service description - example 1

srv	ServiceB
URI	sip://server.com/charging
service_provider	ProviderB
function	audio_transmition
charging_tariff	VoIP
charging_standard	CAPv3
description	Charging service for VoIP

Table 5.4: Service description - example 2

5.2.4.1 Service attributes

Service properties are described as a set of service attributes. A service attribute consists of an attribute identifier and an attribute value. Attributes of a service can be interpreted in terms of resource-based reasoning as resources or functionality provided by a service. For example, an attribute with identifier ”codec” and value ”G.711” can be interpreted to mean that the service described by this attribute provides the functionality of a G.771 codec. Each service description contains a unique identifier, e.g. an attribute srv that contains a value used to uniquely identify the service. In addition, service attributes can be also used for describing general service properties such a human-readable description, the URI of a service, etc.

For the textual representation of service attributes, the LDAP Data Interchange Format (LDIF), described in RFC 2849, is used in this document. LDIF is a widely accepted format for textual data representation and is used by LDAP directories for managing service information.

Table 5.3 and 5.4 are hypothetical examples of service descriptions for a VoIP and a charging service. According to LDIF syntax an attribute identifier is separated from its attribute value by a colon ”:”.

srv	ServiceA
objectclass	SIP_services
objectclass	Voip_services
URI	sip://server.com/voip
service_provider	ProviderC
function	audio_transmition
Audio_codec	GSM
Audio_codec	G.711
description	VoIP service

TABLE 5.5: Service description - example 3

Service attributes can be referenced from service constraints and from application skeletons. Therefore, it is essential for the designer to know, which attributes a service can contain.

Services are organized into service classes that describe the set of attributes for each service. All services within a class have the same set of possible service attributes. Each specific service of a class may have a different set of values for these attributes. A service can belong to multiple classes and as a result can have all attributes of these classes. Examples for service classes can be ”VoIP services”, ”charging services” or ”SIP services”. The classes are defined in a schema. The definition of a schema is inherently fixed, since schema volatility may introduce incompatibilities.

A special service attribute objectclass is used for specifying service classes of a service. This attribute allows referencing classes of a service from service constraints and application skeletons. Listing 5.3 is an example of a schema with two service classes.

```
Schema
Objectclass \gls{sip}\_services {
URI, function , description
}
objectclass voip\_services {
audio\_codec
}
```

LISTING 5.3: Schema with two service classes

Table 5.5 defines a service description derived from both classes of the aforementioned schema:

5.2.4.2 Service constraints

Pre-conditions and post-conditions of a service can be defined via service constraints. Service constraints can be interpreted in terms of resource-based reasoning as resources that are consumed by a service. A service constraint is a logical expression that is formulated by referencing service attributes. The usage of logical, mathematical, string operations and parentheses is allowed in constraints. For the textual representation of service constraints the LDAP Filter format as described in RFC 2254 is used. Listing 5.4 illustrates a service constraint describing the necessity of a charging resource with the charging standard VoIP is required.

```
(&(function = charging)(charging_tariff = VoIP))
```

Listing 5.4: Service constraint example 1

In this example, function and charging_tariff are attribute identifiers and charging and VoIP are required attribute values. This constraint is valid only if a service description exists that accommodates these two requested attribute values. For instance the VoIP service described in the examples in subsection 5.2.4.1 fulfills this constraint.

Instead of service attributes runtime variables can also be referenced in constraints. Runtime variables were discussed in detail in subsection 5.2.3.1. Listing 5.5 contains a constraint that uses a runtime variable to indicate that this particular service cannot be the first service to be executed in a composition.

```
(skeleton.executed_elements_count>1)
```

Listing 5.5: Service constraint example 2

Constraints are grouped into global and local constraints. A global constraint must be valid in the context of a composite service, i.e. each service in a composite service must fulfill the global constraints. Global constraints are useful for defining restrictions applicable to all components of the composite service, e.g. all services in a composite service must be from the same provider or all services must be SIP services.

A local constraint must be valid in the context of the neighboring components of the service, i.e. services connected to the service with the local constraint must

srv	ServiceA
objectclass	SIP_services
objectclass	Voip_services
URI	sip://server.com/voip
service_provider	ProviderC
function	audio_transmition
Audio_codec	GSM
Audio_codec	G.711
description	VoIP service
constraint	($(function=charging)(tariff=VoIP)(provider=providerC))

Table 5.6: Service description with local constraint - example 4

fulfill this constraint. In contrast to global constraints it is not necessary that all neighboring services fulfill the local constraint. In this case it suffices if at least one neighboring service fulfills it. Consequently a service describes what services it is compatible with through its local constraints.

Constraints are part of the service description and will be defined using special service attributes: a global constraint through service attribute global_constraint and a local constraint through constraint service attribute. Table 5.6 is an example of a VoIP services that requires a charging service from same provider as VoIP and with the charging tariff VoIP:

Figure 5.9 illustrates the impact areas of local and global constraints.

Multiple global_constraint attributes in a single service description are joined with a Boolean AND "&" into one global constraint. In contrast multiple constraint service attributes are not connected and will be considered as two independent constraints. It is therefore possible that one local constraint will be satisfied by one service and another local constraint by another service.

Table 5.7 illustrates a service requiring that at least one of its neighboring services should provide the function charging for tariff VoIP and least one of its neighboring services should belong to providerA. However these constraints can be fulfilled by two different services.

Table 5.8 is an example of a service with a global constraint that forces all services in the composite service to be SIP services offered by providerA.

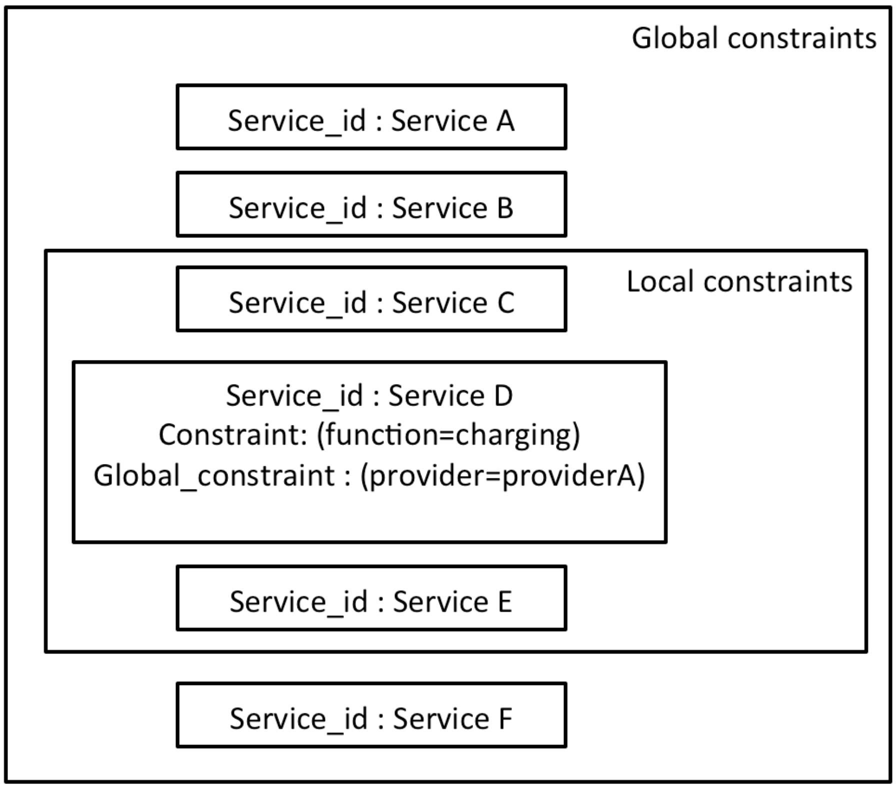

FIGURE 5.9: Local vs global constraints

srv	ServiceA
objectclass	SIP_services
objectclass	Voip_services
URI	sip://server.com/voip
service_provider	ProviderA
function	audio_transmition
Audio_codec	GSM
Audio_codec	G.711
description	VoIP service
constraint	(&(function=charging)(tariff=VoIP))
constraint	(provider=providerA)

TABLE 5.7: Service description with local constraint - example 5

srv	ServiceA
objectclass	SIP_services
objectclass	Voip_services
URI	sip://server.com/voip
service_provider	ProviderC
function	audio_transmition
Audio_codec	GSM
Audio_codec	G.711
description	VoIP service
constraint	(&(function=charging)(tariff=VoIP))
global_constraint	(&(service_class=sip_services)(provider=providerA))

TABLE 5.8: Service description with local and global constraint - example 6

From a complexity theory point of view constraint resolution is defined as a satisfiabillity problem whose instance is a Boolean expression written using only AND, OR, NOT, variables and parenthesis. Boolean satisfiability problem are interesting because it is an NP-Complete problem and therefore incorporating this feature into our system, does not yield to poor performance as we examine later on in the chapter 6 where we evaluate our prototype work.

5.2.5 Execution Agents

This section describes the implementation details of two execution agents, the SIP Execution Agent and the HTTP Execution Agent. We will begin our description with the SIP Execution Agent since its implementation is far from trivial.

5.2.5.1 SIP Execution Agent

This subsection describes the implementation of the SIP EA and presents one example use case with signaling flow. The basic routing mechanisms include routing in the SIP/IP core S-CSCF, e.g. triggering of messages to JSR116 containers. Furthermore, for the purposes of invoking SIP servlets, routing/dispatching takes place within the container. As always, reducing signaling across different network elements is desirable. With that in mind, the working assumption is that most routing takes place on top of the container/dispatcher, where most enabling services are invoked and potentially also the communication service (e.g. chat). This is achieved by keeping most signaling within the application server (AS), i.e. either via the JSR116 dispatcher or internal in the servlets. This reduces the involvement of the S-CSCF and the less signaling towards the S-CSCF, the better.

Routing outside of the container e.g. to address a communication service deployed on a different AS is performed under the assumption that the request will terminate outside the initial AS. In case the request leaves the original AS two alternatives exist:

- Either the request can directly terminate at the next application server,
- or it passes via one or several other intermediate application servers.

The later case can only guarantee consistency in the execution of the composition, when all involved application servers share state with the same composition core.

As mentioned, there are some architectural alternatives when it comes to the placement of communication service components such as chat and VoIP, as well as what addressing mechanisms are used. Two main alternatives are identified:

1. Composition logic, enabling services and communication services deployed as SIP servlets on different application server (split) or on the same AS. This could imply one, potentially distributed container or different containers.
2. Composition logic and enabling services deployed as SIP servlets on an AS and communication services separately deployed as UACs.

Alternative 1 (Figure 5.10) assumes the communication service endpoints, (e.g. chat and VoIP) to be deployed on the JSR116 SIP container as SIP servlets. They are reachable over the ISC interface via triggers set in the S-CSCF.

Regardless of how the dispatching is done at the container level, there is a first routing step on the CSCF level, in order to trigger traffic to the appropriate container. Taking a chat conference as an example, the trigger setting needs to be able to handle both the messages going to the conference factory during conference creation and messages' going to an individual conference as a user joins a conference (dial-in). The trigger that is set in the S-CSCF is first-come first serve (FFS), but alternatives include:

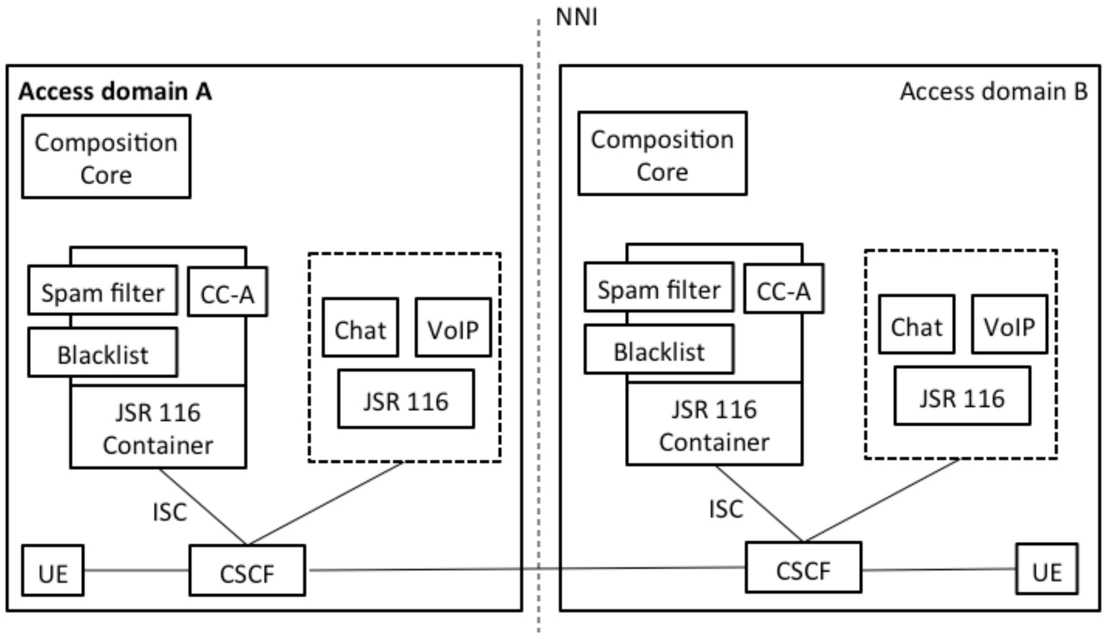

Figure 5.10: Composition logic and communication services on different application servers

- trigger on part of URI, e.g. "conf*" in the To: header field of the INVITE
- trigger on Communicaton Service feature tags, e.g. accept-contact: +g.oma.sip.im

The iFC triggers cause the SIP INVITE to be routed up to the JSR116 container, where further dispatching occurs potentially involving composition logic.

If there is split functionality (i.e. composition on one AS and chat functionality on another AS), the dispatching in the CSCF must make sure that the application servers are invoked in the proper order, in accordance with the proper S-CSCF procedures.

- One way would be prioritization of triggers (e.g. based on feature tags) in the S-CSCF. The MSP prototype work will use a generic feature tag, "ER-EAS" to route to the container. However, in order to avoid implementation dependencies, this requirement is not supported. The deployment structure of the prototype is also assuming one common AS for both chat and VoIP services.

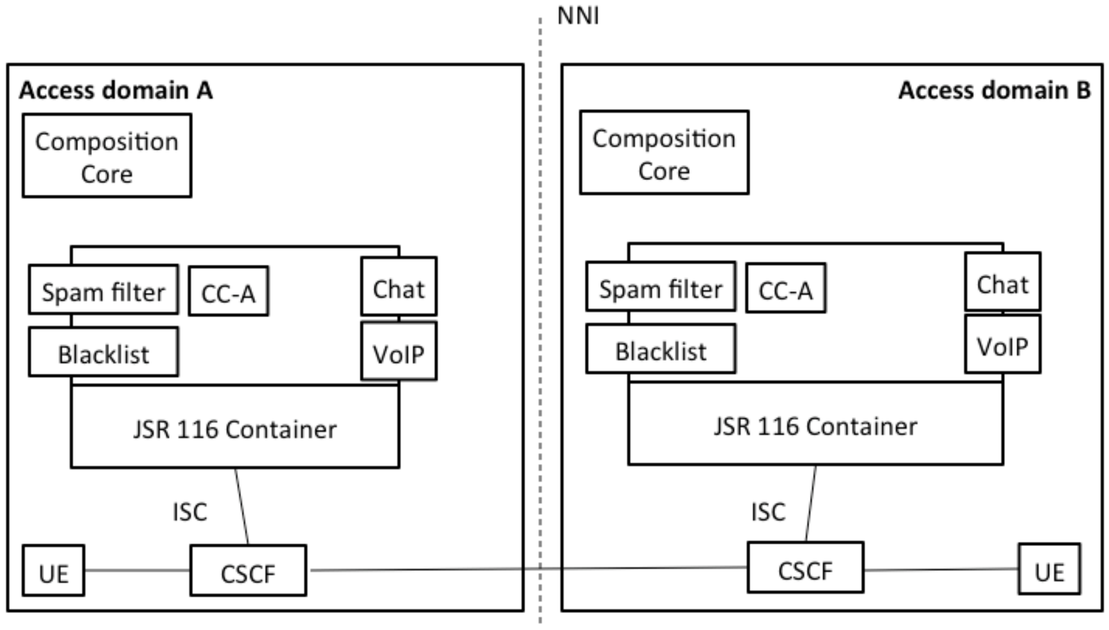

FIGURE 5.11: Composition logic and communication services on the same application server

- Another way would require visited ASs to make modifications (e.g. additional header fields) to the request that is parsed by the S-CSCF, so that it knows what AS has been visited without having to keep the state.

In JSR116 several alternatives exist for manipulating the body, header fields and parameters in the SIP requests so this can be done in several ways within the container.

Alternative 2 (Figure 5.11). This approach is based on registering communication services as UACs in the S-CSCF registrar as any other client. This makes them directly routable, using the PSI, but has the major drawback in that every new conference needs to register itself in the S-CSCF. It is a similar process as when the new conference is a SIP servlet and registers in the container. However, reducing the involvement of the S-CSCF is of major importance. Further, it is not adhering to current standardization activity in OMA, where the OMA TS on specifies that the chat service endpoint shall reside on top of the ISC interface in the case the SIP/IP core corresponds to IMS. Therefore, this alternative will not be considered in the architecture.

5.2.5.2 Way forward for prototype work

For deployment of the services on one or several ASs (alternative 1 and 2), there is another choice to make. That is whether or not new conference instantiations shall become servlets of their own or if they shall be internal to the chat/VoIP servlet.

If each new conference instantiation will become a stand-alone servlet, they need to register themselves in the JSR116 container in order to be routable by the dispatching functionality of the container. The container may then use the full URI in the dispatching process.

To avoid the registration procedure during runtime and the impact on the container, the choice for the prototype has been to make the conferences internal to the chat servlet.

In our prototype implementation chat and VoIP functionality are deployed as SIP servlets on top of the JSR116 container. Each new conference will not create a new servlet. Instead, they will be internal to the chat servlet. This means that the chat servlet will handle the dispatching to each individual conference.

Advantages of this approach are:

1. Reduced interaction with S-CSCF during runtime (as opposed to the case of individual conference servlets in AS, or the UAC case)
2. Reduced interaction with JSR116 container during runtime (as opposed to the case of individual conference servlets which would need to register)

Our prototype uses two addressing mechanisms:

1. Generic Application Service Communication Service feature tag triggers all traffic to the SIP container. This is a feature of the prototype work and test-bed configuration and is not the approach suggested.
2. Container routes traffic to the chat servlet based on the chat feature tag
3. Chat servlet routes internally to the correct conference alt. conference factory, based on the URI/PSI.

5.2.5.3 SIP routing in the container

As described in 4.3.5, the main entity dealing with composition execution on a SIP container and therefore main handler of the dispatching chain and related SIP signaling is the SIP Execution Agent. On a JSR116 compliant SIP container, it would be therefore the natural choice to implement a SIP Execution Agent as a SIP-proxy and make it responsible for the routing of SIP messages between servlets. Fortunately, JSR116 specification provides APIs allowing for implementation of such functionality.

The JSR116 specifies APIs for processing of SIP messages within a compliant container. Amongst other things these APIs provide the possibility for manipulating SIP requests. Within the context of this work we are particularly interested in functionality that can be used to control routing of requests inside the container, because this provide the means for controlling the order of SIP servlets invocations. The JSR116 API defines such functionality based on RFC3261 as part of the SIP-proxy servlet. SIP proxies' route SIP requests and responses. A request may traverse several proxies on its way to a UAS. Each will make routing decisions, modifying the request before forwarding it to the next element. Responses will route through the same set of proxies traversed by the request in the reverse order.

5.2.5.4 JSR116 API related to routing

This section describes methods provided by the SIPServletRequest Interface that enable the modification of the proxy parameters before proxying SIP requests.

RFC3261 specifies it is possible for a proxy (the SIP Execution Agent in our architecture) to mandate that a request visit a specific set of proxies before being delivered to the destination (section 16.6 of the RFC). This set of proxies is represented by a set of URIs. This set MUST be pushed into the Route header field ahead of any existing values, if present. If the Route header field is absent, it MUST be added, containing that list of URIs.

By modifying proxy parameters such as the headers and the body and specifically the Route header field, the route the SIP messages will take can be determined. In our prototype this will be the responsibility of the SIP Execution Agent.

The pushRoute method of the SIPServeletRequest interface defined in the API of JSR116 allows the manipulation of this header field:

```
public void pushRoute(SipAddress proxyAddr);
public void pushRoute(SipURI proxyAddr);
```

This can force the SIP message to visit specific proxies before being delivered to its destination. The pushRoute methods effectively adds a route header field to the SIP request ahead of any existing Route header fields; if there are no Route header fields then this method will add one.

In order to ensure that an application stays on the return signaling path, it is possible to add a servlet to the record route:

```
Proxy p = req.getProxy();
p.setRecordRoute(true);
```

This can be done even without affecting the request URI. This effectively creates a reverse dispatching chain that contains all servlets that need to receive return signaling.

JSR116 defines a specific mechanism for proxy functionality in the interface javax.servlet.sip.Proxy. If it is necessary to redirect a request to only one proxy you may not need to use the pushRoute method. It is possible to use simply an ordinary proxyTo API-method. However this does not prevent the downstream proxy from performing another proxyTo:

```
public void proxyTo(URI uri);
public void proxyTo(java.util.List uris);
```

The argument identifies a proxy (or list of proxies) that should be visited before the request reaches it final destination. If an application generates its own final response it cannot proxy to more destinations.

The discussed APIs provide the functionality to intercept, reroute and even change the actual content of SIP messages. However, they should be used with great caution. In theory, application servers are not supposed to be aware of other application servers, or at least should not be created that way. However, extensions to basic SIP routing functionality require knowledge in the application logic further downstream.

Consequently, changes to SIP messages that require additionally functionality in other application servers downstream are very difficult to implement reliably. Functionality based on such extensions cannot be guaranteed to be executed correctly, or even be executed at all, as it depends on the interpretation of the SIP message by the AS.

5.2.5.5 Use case description

This section illustrates how composition logic is applied to a simplified use case, where only two administrative domains are assumed; one operator network with composition logic and one without this logic.

The scenario is that user A creates a conference and asks user B to join. The conference server is deployed in user A's network, as well as servlets for Blacklist and composition logic.

The main steps are illustrated in sequence diagrams in Figures 5.12, 5.13 and 5.15. A description of each step follows: (TCP procedures are omitted)

1. In order to create a conference instance, User A sends an INVITE(1) to the conference factory

```
INVITE(1) To: sip:conference_factory@servcer.com
Accept-contact +g.oma.sip.im
```

2. The application server hosting the composition logic is invoked via iFC. The composition logic determines that a blacklist check should be made as a first step. The route for the request is modified via the PushRoute method, where the blacklist servlet is added. Further, the SIP Execution Agent adds itself next in the route list, in order to be able to make further evaluations. If the SIP Execution Agent determines that no further composition logic is needed, it may at this point also set the full dispatching chain (including chat) and does not then include itself. However, in this example we show this in two steps.

```
200 OK (2)
To: sip:a@domainA.com
From: sip:conferencefactory@domainA.com
Contact: sip:conference123@server@domainA.com
```

3. Composition logic makes sure that a Blacklist check is made on whether or not User B is allowed to access the conference resource and create a conference. The check is successful. INVITE is forwarded back to the JSR116 container. Had the check been unsuccessful, then the request would have terminated in the blacklist servlet and an error response would have been returned. The servlets act as stateless proxies.

4. The INVITE is forwarded back to the container and routed on to the SIP Execution Agent.

5. The SIP Execution Agent and the composition core determine that the next step is to send the request to the endpoint, which in this example is the chat servlet and more specifically the conference factory within the servlet.

6. The conference factory creates the conference instance and returns the conference instance URI in the 200 OK response to User A. User A has now joined the conference.

7. User A instructs user B to join the conference instance by sending a REFER message. FFS what composition logic is applied to the REFER.

```
Refer(3)
To: sip:a@domainA.com
From: sip:a@domainA.org
Contact: sip:a@domainA.org
Refer-To: conference123@server.domainA.com; isFocus; method=
    INVITE; text
Referred-by: sip:a@domainA.com
Refer-sub: false
Supported: norefersub
Body:
<recipient-list-url>
<entity uri="sip:a@domainA.com"/>
<entity uri="sip.b@domainB.com"/>
<entity uri="sip.c@domainC.com"/>
</reciepient-list-url>
```

8. User B decides to join and sends an INVITE to the conference instance within the chat servlet.

```
INVITE(4)
```

```
To:  sip:conference123@server.domainA.com
From:  sip:b@domain.com
Referred-by: sip.a@domainA.com
```

9. A dialogue is established between user B and the conference, i.e. he joins the conference.

In addition to the above-mentioned steps, notification packages (subscribe/notify) are set up to inform users in the conference of events (e.g. joining/leaving users). In the example, these are assumed not to invoke the composition logic.

In this use case example, there is no branching other than the implicit one as a direct result of the blacklist check. However, if a service like a busy check is implemented, the result of the servlet action might result in a branching decision to e.g. either go ahead with the session setup or divert to a mailbox service.

5.2.5.6 Correlation for conferences

The correlation header named Same-session that has been proposed in the IETF intends to facilitate the correlation of SIP dialogues. In a peer-to-peer example, two users might be involved in a chat and decide to add a voice component. The sender of the SIP INVITE for the voice component can then choose to populate the Same-session header with information regarding the SIP dialogue that was used for the ongoing Peer-to-Peer (P2P) chat. This would allow the other party to know that these two SIP dialogues are correlated, and the receiver could e.g. determine that they should be part of the same Graphical User Interface (GUI) and conversation tab in a myTalk application. Now, for a conference scenario the situation is somewhat more complex. Assuming that each participant in a conference has their SIP dialogue and Message Session Relay Protocol (MSRP) connection to a chat focal point and Media Resource Function Processor (MRFP) respectively, this information is shared between the two ends. Thus, when one conference participant wants to upgrade the chat conference to a chat+voice conference and sends a REFER message to the others telling them to join a VoIP conference with a certain URI, there is no way for him to populate the Same-session header with the information needed to correlate the new SIP dialogue for VoIP with the existing chat dialogue of the receiver. This

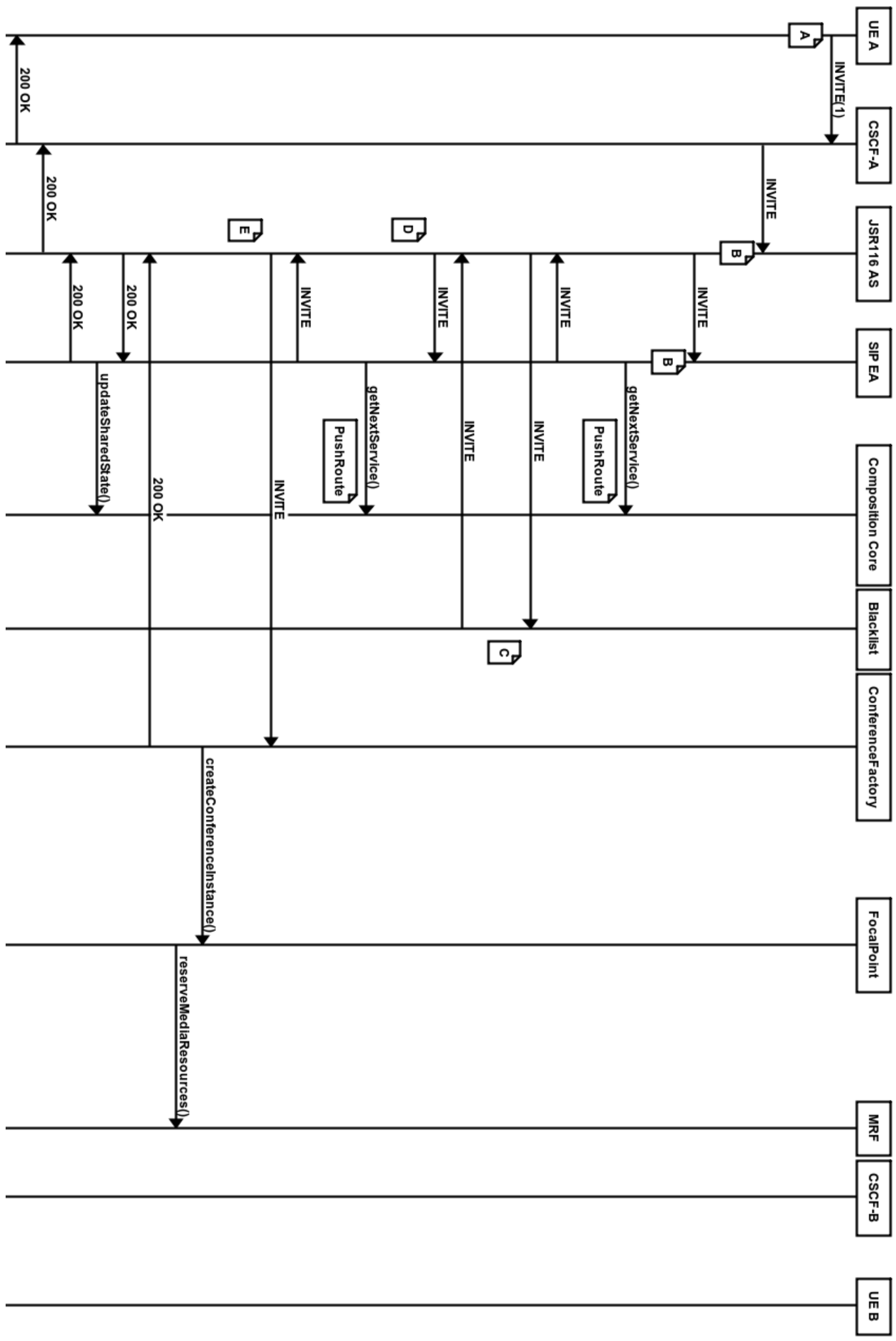

Figure 5.12: SIP-EA / Composition core intraction, Steps A-E

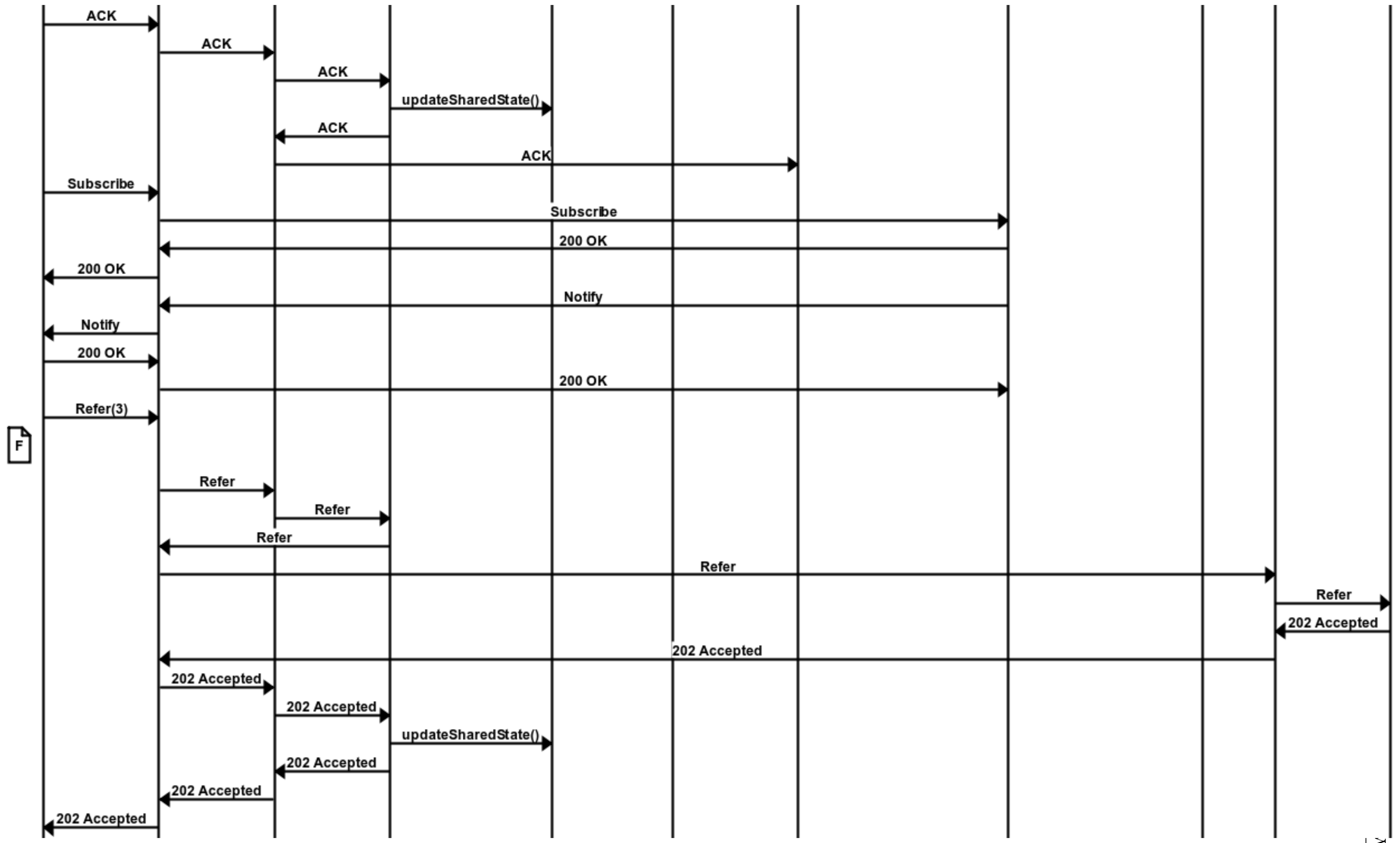

FIGURE 5.13: SIP-EA / Composition core intraction, Step F

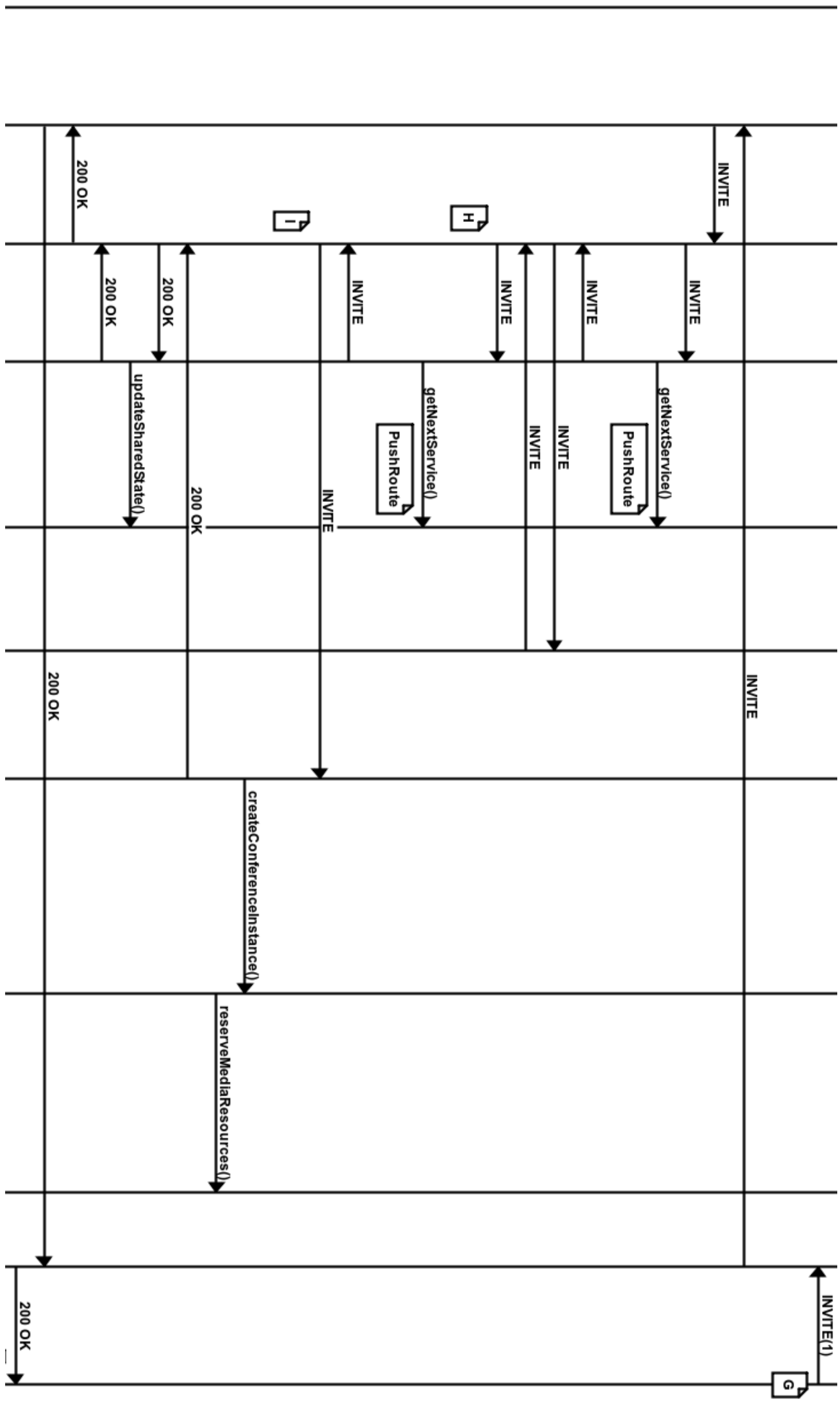

FIGURE 5.14: SIP-EA / Composition core intraction, Steps G-I

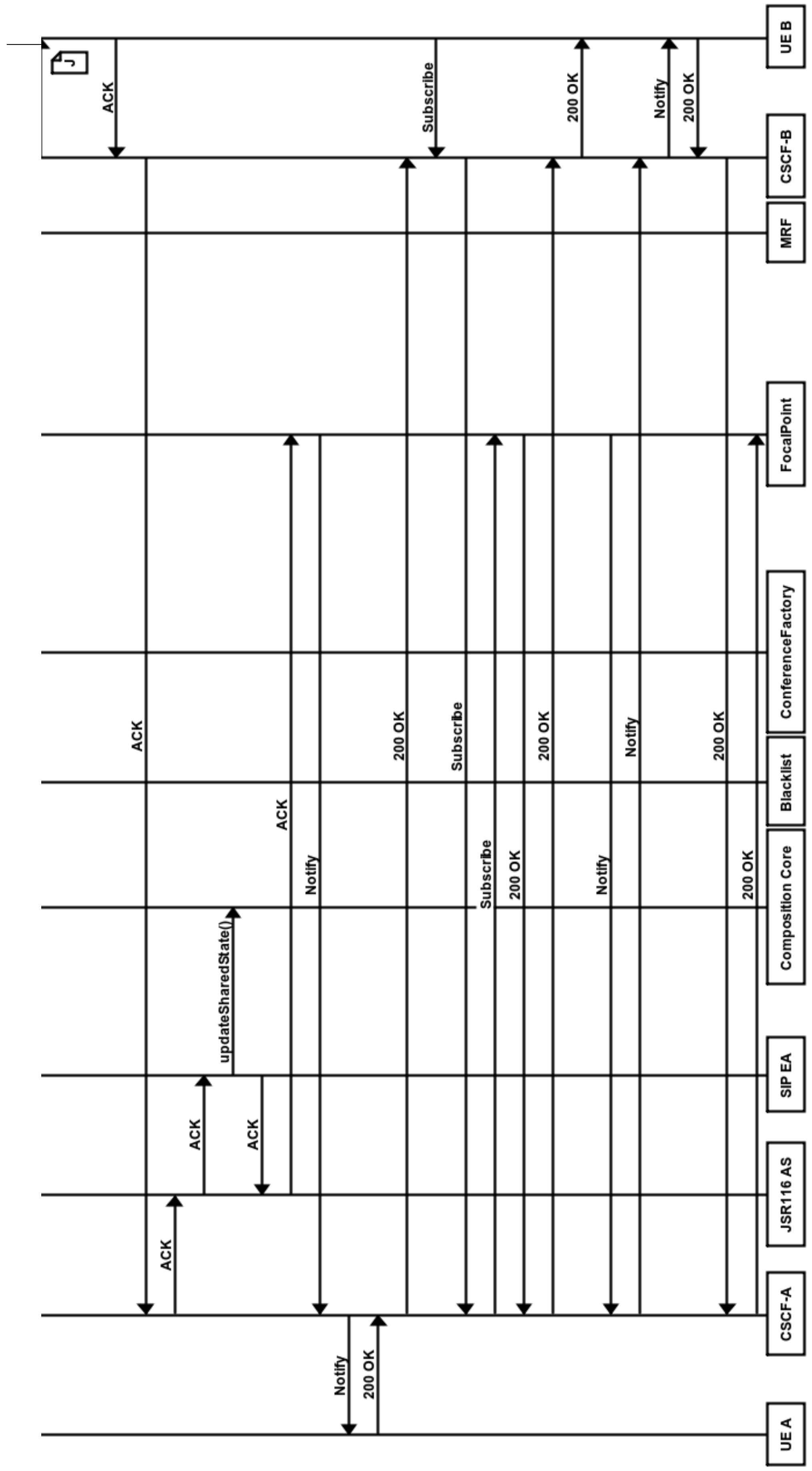

Figure 5.15: SIP-EA / Composition core intraction, Step J

happens, since he does not know the Call-ID of the dialogue between the other user and the chat focal point.

The following three alternatives are identified:

1. SIP URI: One way to indicate correlation could be to allow the Same-session header to indicate a URI. At present, the Same-Session header identifies a dialog using the Call-ID, From and To tags. This is in fact the correct way to identify correctly and uniquely a dialog in SIP. However, in the conference scenario, because each participant has just one dialog with a specific focal point, the URI of the chat conference should be enough to identify the right dialog. The URI of the chat conference is information known to all participants in the conference and that is a unique identifier for the chat conference.

   The receiver would then know that the INVITE that he will send to the VoIP conference (as a result of the REFER), will created a dialogue that is correlated to whatever dialogue he currently has with the end-point corresponding to the URI indicated in the Same-session header. This information should be sufficient since there can only be one dialogue connected to each focal point URI.

2. Event Package for Conference State: Another way to learn the information needed to correlate the new SIP dialogue for VoIP with the existing chat dialogue of the receiver is implementing and using the Event Package for Conference State draft (draft-ietf-sipping-conference-package-12.txt). This package provides to the participant the needed information to use the Same-Session header in the way specified in the correlation draft. The drawbacks of the alternative are the effort necessary to implement the Event Package and the amount of the information exchanged among the participants especially on the wireless link.

3. Using the chat to exchange the dialog information: A participant could use the MSRP chat or a Message method in order to learn the information need to upgrade the chat conference to a chat+voice conference. A simple request-response message protocol where a participant asks by message the dialog information to the other participant and the other provides it. There are not any security issues because all the participants know who are the users involved in the conference and they will reply only to those users.

5.2.5.7 Specification and Implementation

A SIP container making use of the composition core must have a SIP Execution Agent deployed on it. This entity in co-operation with the composition core is responsible for the execution and control of the composition on this particular SIP container. Communication between the c and SIP Execution Agent cannot be based on the SIP protocol since a richer semantic is required; instead common middleware technologies, e.g. WS, J2EE RMI, etc. would be more appropriate.

When the SIP Execution Agent is invoked, it typically executes three tasks:

- it informs the composition core about the current state of the SIP session,
- it receives instructions from the composition core
- it updates the dispatching chain of the SIP-container

These steps break up the static mechanism of SIP servlets composition creation as defined in JSR116 and help implement this new and very dynamic approach.

5.2.5.8 Possible SIP composition execution realizations

The SIP servlet container and specifically its dispatcher present a very good basis for implementing more advanced composition functionality for SIP service composition. The task of the extended dispatcher would be to cater for non-sequential execution of servlets, thereby executing the composition defined by the composition core. There two major possibilities for implementing this approach:

- Extended SIP Container Servlet Dispatcher — If there is a standardized method of extending the application/servlet selection and routing functionality, then the SIP Execution Agent can be implemented in a portable way using this method. Currently, the JSR116 specification does not provide such an opportunity. But the upcoming JSR289 specification could include such a feature. One of the extensions proposed for this specification by AT&T in May 2006 describes an API and further mechanisms for specification of developer defined "application router" objects that essentially extend a container with a application/servlet selection and routing functionality. It is explicitly stated, that such an ”application router” can use information coming from some external sources for making its decisions. However, it is not the intention of the proposal to concretely specify how this happens. Instead, only mechanisms and interfaces for interactions between application routers and SIP containers are standardized. Obviously, if this kind of proposal is standardized as JSR289, the implementation of a SIP Execution Agent on top of such a container would become a rather easy task.

- Dedicated SIP Servlet — A different approach that follows a similar schema yet offers considerably more flexibility is to employ a dedicated SIP servlet responsible for controlling the composition execution. This approach in principle requires no modification to the SIP container, as the dedicated servlet would be deployable on any container and be executable by a standard dispatcher as part of the dispatching chain.

  The dedicated composition servlet should always be unconditionally invoked first as a response to a SIP request for a composition. This servlet does not necessarily need to implement any application specific logic itself, even though in special cases such implementation would be useful. Instead, it takes over the role of the dispatcher and invokes other servlets deployed on the same (or possibly other) containers.

  Additionally a special servlet is always executed as the last one in the chain might be required. This servlet would be configured as a default outgoing proxy for the container. By doing it, all outgoing messages are passing this servlet and there is no possibility for a SIP packet to leave the container without notice.

  This pair of first and last servlets has a full control over SIP packets flow inside a chain of servlets.

  Such a dedicated SIP servlet is likely to be somewhat different from normal servlets because it should be able to enforce the execution and control of other servlets. Implementations of such a servlet may require some sort of access to the lower-level capabilities of the SIP container, which are usually not exposed via standardized APIs (e.g. not specified in JSR116). Therefore, it might need some extensions of a SIP container specification. One of the JSR289 features proposals from IBM even suggested something along these lines. Namely, a special composer servlet was proposed. Such a composer servlet is treated in a special way and even should be even deployed in a special directory.

For the purposes of this prototype, we decided to use the Dedicated servlet-based approach, due to its portability.

5.2.5.9 SIP Execution Agent servlet specification

High-level design The goal of a SIP Execution Agent is to enforce the execution of composition decisions on a SIP container by influencing the selection and order of execution of the applications and servlets participating in a processing of messages related to a given SIP session.

Based on some logic, the SIP Execution Agent decides which SIP servlet is to be executed next. When this decision is taken, it instructs the SIP container to execute the selected servlet. After the execution of the servlet, the control returns back to the SIP Execution Agent.

It should be obvious from the description that a SIP Execution Agent has a possibility to influence the order of servlet execution after any step. Essentially, it introduces a dynamically defined workflow of execution at the SIP container level. This is a great step forward compared to static linear workflows possible with a standard SIP container and goes beyond approaches similar to Bea's. These approaches made their composition decisions either statically at deployment time (see Table 4) or statically at runtime by means of a special first servlet (see Table 5). In both cases, the chain of servlets to be executed was decided at once and could not be changed afterwards. With the proposed SIP Execution Agent approach, the sequence and order of servlets to be executed by a SIP container is decided on a step-by-step basis (see Table 6).

As described above, it is open whether a SIP Execution Agent is making its composition decisions using its local knowledge or consults and external node. In some cases, e.g. when the whole composition resides on a container where SIP Execution Agent is running, it is enough to use just local information. However, in a situation where a multi-service composition spans over multiple SIP containers, it could be very desirable to align and to plan the decisions made by SIP Execution Agent

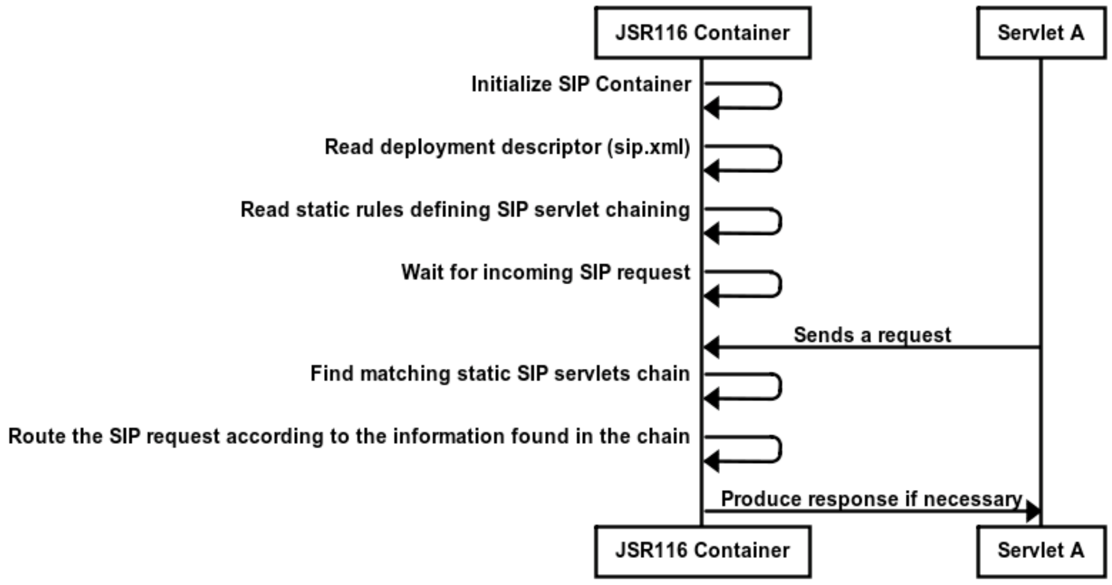
JSR116 Container
Servlet A
Initialize SIP Container
Read deployment descriptor (sip.xml)
Read static rules defining SIP servlet chaining
Wait for incoming SIP request
Sends a request
Find matching static SIP servlets chain
Route the SIP request according to the information found in the chain
Produce response if necessary
JSR116 Container
Servlet A

FIGURE 5.16: JSR116 compliant container with static servlet chaining

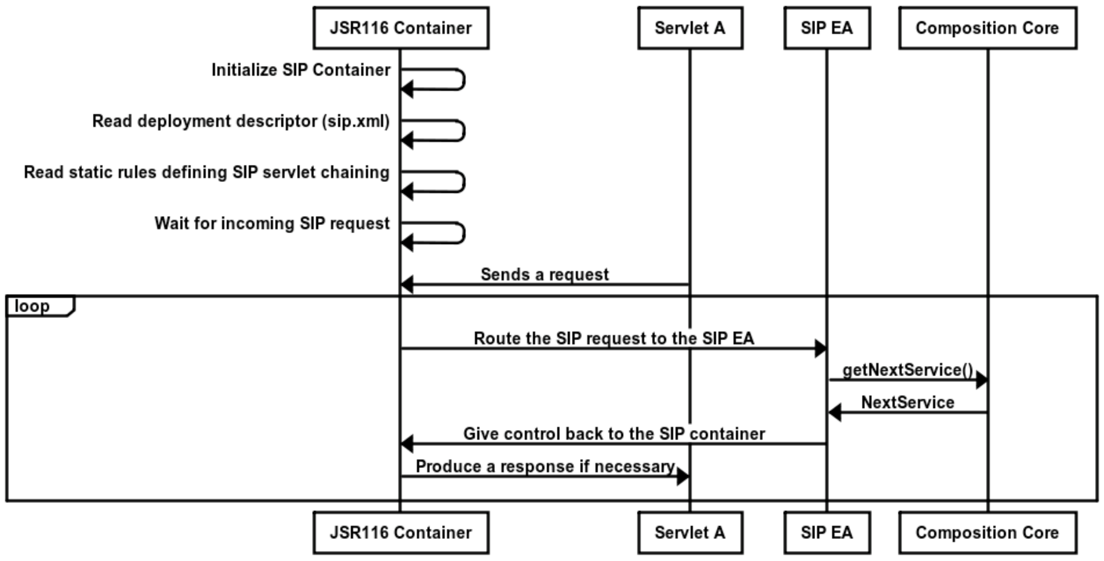
JSR116 Container
Servlet A
SIP EA
Composition Core
Initialize SIP Container
Read deployment descriptor (sip.xml)
Read static rules defining SIP servlet chaining
Wait for incoming SIP request
Sends a request
loop
Route the SIP request to the SIP EA
getNextService()
NextService
Give control back to the SIP container
Produce a response if necessary
JSR116 Container
Servlet A
SIP EA
Composition Core

FIGURE 5.17: JSR116 compliant semi-dynamic servlet chaining

SIP Execution Agent with other participants of this composition. Based on this consideration, it seems appropriate to implement the SIP Execution Agent in such a way, that it would interact with external nodes controlling current composition (e.g. with composition core) and use the provided input when making the decisions for the container where a given SIP Execution Agent is running.

As composition core should be aware of SIP servlets currently available on the execution nodes participating in a composition, one of the SIP Execution Agent functionalities should be able to provide composition core with a list of all available SIP servlets and applications deployed on a given SIP container. This set should include a set of unique identifiers (preferably unique SIP routes to any particular servlet, though some other options can be considered as well) and also references to the descriptions of these servlets in a format understandable by the composition core.

5.2.5.10 JSR116 based implementation of SIP Execution Agent

First of all, at the design/deployment time, it should be ensured that SIP.xml file contains a reference to the SIP Execution Agent servlet in a description of a given SIP application. By doing so, it is guaranteed that SIP container invokes the SIP Execution Agent in the scope of a current SIP session. It is proposed to put SIP Execution Agent as a last (optional) element in the chain of servlets related to a given SIP application described by the SIP.xml file. This allows some servlets to be executed before SIP Execution Agent - ”outside” the composition control, so to say. This makes sense, since after SIP Execution Agent will be invoked, it will take over the control over servlets execution for a given session.

Change in the SIP.xml file is the only one required at the design/deployment time. All other changes in container behavior happen at the runtime, namely, when the application containing a reference to SIP Execution Agent in its specification in SIP.xml file is selected for execution. In this case, SIP Execution Agent servlet is reached and starts controlling further invocations of servlets in scope of a current session. More precisely, SIP Execution Agent performs the following (1-5) sequence of steps:

1. When a SIP Execution Agent is invoked, it will make a decision about which servlet is to be executed next. This decision is based on the information provided by external composition control nodes (e.g. composition core).
2. When the composition decision for a next step is obtained, SIP Execution Agent will enforce that the given servlet is executed as next and that SIP Execution Agent gets the control after it again. In case of JSR116, this is achieved via quite legal SIP Servlet APIs calls, namely via pushRoute() functionality. SIP

Execution Agent simply pushes a SIP URI of the next servlet on the route. Then it also adds itself to the route, so that the control comes back. After the route is modified in this way, SIP Execution Agent simply use a usual send() API method of the SIPServletRequest object, which gives the delivery control to the standard SIP Servlet dispatcher of the SIP container.

3. Dispatcher will determine that this is a local route and deliver the SIP message to the required servlet. Then next servlet's address added in step 2 will be removed from the route.

4. The required servlet processes the SIP message

5. The dispatcher gets a control again. It determines that the next destination point is specified via a local route and that the next element on the route is SIP Execution Agent. Therefore SIP Execution Agent is invoked and the process of selecting the next servlet starts again at the step 1

5.2.6 HTTP Execution Agent

The HTTP Execution Agent (HTTP EA) receives as input a service description for an external HTTP service, along with a set of input parameters and makes an HTTP request towards that service. As such it provides full support for HTTP and therefore it is capable of making HTTP POST, PUT, DELETE and GET requests. This information is defined in the service description for HTTP services. In addition, in the same service description, one can define the endpoint of the HTTP service, specific headers to be included in the HTTP request and last but not least the body of the HTTP request. This set of parameters allows for complete support for every RESTful invocation.

A special case of an HTTP service is SOAP based service also known as Web service. That is because SOAP based services, at least in their simplest form which does not require the existence of a SOAP stack such as the one provide by JAX-WS [91], can be viewed like any other RESTful services with the exception they are defined as HTTP POST requests, with a Content-Type of ”application/soap+xml” and a body that complies to the format of SOAP.

An example of such a body is shown in listing 5.6:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>${company_name}</m:StockName>
  </m:GetStockPrice>
</soap:Body>
</soap:Envelope>
```

Listing 5.6: Annotated HTTP request payload

Input parameters can be provided as runtime variables that later on can be mapped to the query string for RESTful services (i.e. http://url.com/query_parameter1=val1&query_parameter2 and to specific placeholders in the body of the HTTP request (notice the special notation ${company_name} in the previous snippet).

There are three different variants of HTTP requests that are supported.

The first variant is request only: this entails that an HTTP request is being made and the invocation thread is immediately released after that without waiting for the response from the external service.

The second variant is request-and-response: this entails that an HTTP request is being made and the invocation thread is blocked until the external service returns a response which later on converted to a set of runtime variables in order to become available to the application skeleton.

The third variant is non-blocking request-and-response: this entails that an HTTP request is being and the invocation thread is not blocked. However, as soon as the response from the external service is received, the particular session in the shared state manager is updated to include the runtime variables that are generated from that response.

An interesting feature available in our prototype, allows for the automatic generation of a service description from a WSDL description, due to the formal description available. RESTful service descriptions, similarly to SIP service description need to be crafted manually. Even though WADL exists, we decided not to include it in our prototype, for the purposes of automatically generating a service description since WADL is not as widely adopted as WSDL.

The underlying implementation of the HTTP EA is achieved by using the async-http-client [81] so essentially the HTTP EA acts as a wrapper [61] to this external library.

5.3 Composition framework

This section describes in detail the functionality of the composition core. More specifically, the first section is dedicated to the integration between the composition core and the rest of the components, comprising the composition framework. The second section focuses on the composition core itself and describes and describes its internal structures and functionalities.

5.3.1 Composition framework component interaction

The composition core implements the creation of composite services by determining the structure of the composite service and defining appropriate services. The composition core creates a composite service based on static information, i.e. application skeleton and services descriptions, and runtime information, e.g. user preferences, user request, resources allocation. A composite service is created element-by-element, so that every service in a composite service will be not determined until the previous service in the composition is executed. This allows taking into account changes in the runtime environment that could occur at runtime. It should be noted that the composition core does not directly invoke services, rather the composition core suggests to the EA, which services to be executed next, in order to create the requested composite service. This process consists of four main steps:

1. Invocation
   - EA communicates with the composition core the composition core API
   - Parameters provide information on:
     - the external request (i.e. SIP packet, HTTP request)

    - additional related state information
    - reference to the composition session

2. State evaluation

   - composition core evaluates the current state of the composition session
   - Identifies the current state in the composite service
   - Identifies "loose ends" suitable for extending the composite service
     - not terminated SIP sessions
     - possibility to establish new chains

3. Query evaluation

   - composition core evaluates all queries in the composition in order to narrow down the amount of compatible services
     - Composition-wide evaluation is necessary to cover feature interaction across SIP sessions.
   - Composition core determines the next to be executed service.

4. Composition decision

   - Composition core returns the service to execute next.

After the execution of the indicated service the execution agents invokes contacts the composition core again and either asks for the next service in the SIP service chain or for an alternative service in case the suggested service could not be executed successfully. In order to prevent resource collision between services, the composition core must ensure that all existing composite services within the overall user composition session will stay valid and that all related queries are evaluated before a service is suggested for execution.

The composition core and the Shared State Manager handle management of runtime information and composition sessions jointly. Figure 5.18 illustrates in UML sequence diagram notation the four-step process of composite service interpretation.

A more detailed description of the process illustrated by an UML activity diagram is given in section 5.3.1.1.

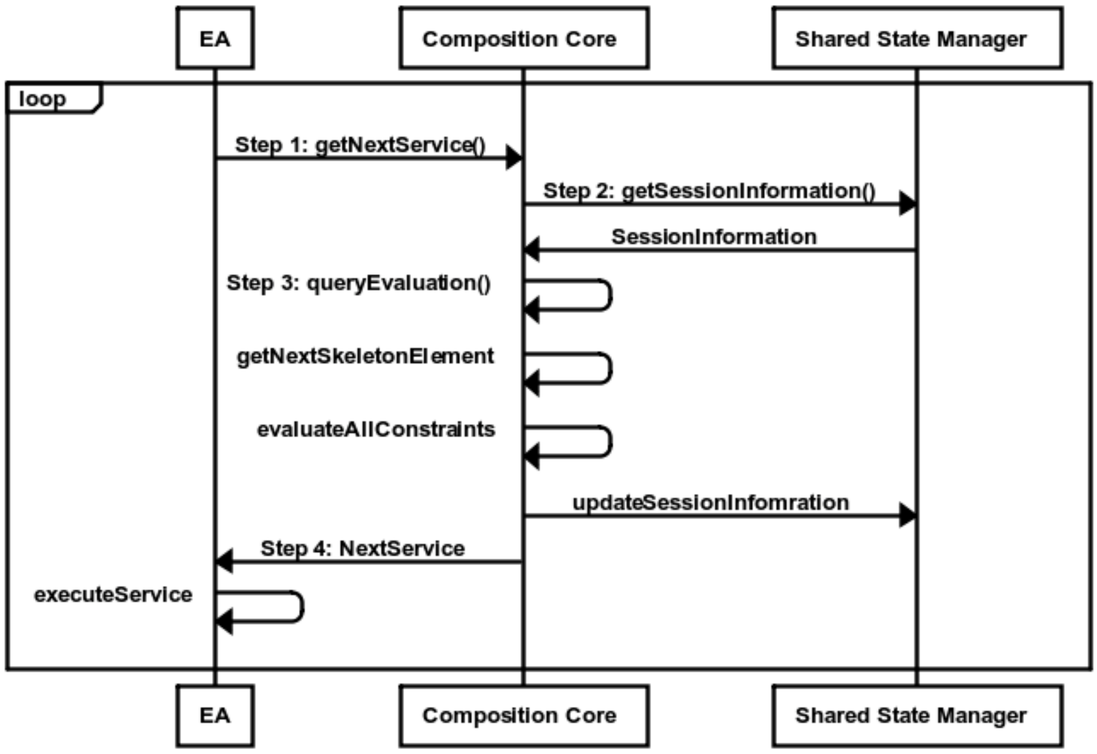

FIGURE 5.18: Interpretation phases

Depending on the number of services in a service composition and the number of running service compositions some optimizations in this cycle can be applied. One possible optimization is to reduce the number of loop interactions and the number of requests from EA to the composition core. This can be done by returning bundles of alternative services, so that the composition core provides to the EA not only the immediate next service, but provided that the application skeleton allows this, two or more next services at a time. This is achievable only if there are services in the application skeleton whose selection is not depended from the results of previous selections or other runtime information. However, the most time intensive task of the actual composition core internal processes is the evaluation of queries. This task can be optimized through pre-compilation of queries, so that only queries, that have references to the attributes of new services, will be evaluated.

5.3.1.1 Composition core activity diagram

When triggered by the execution agent the composition core will first try to identify an existing session in the shared state manager that corresponds to the incoming request (Activity 1 in Figure 5.19). If such a session does not exist, a new session will be created (Activity 2). For the creation of a new session, the composition core needs to know which application skeleton should be used.

After a new session is created, we collect the list of actions that can be interpreted based on their data dependencies (Activity 3). If the composition that corresponded to the request already exists, the status of last service execution within this composition will be proved based on runtime information (Activity 4). If the atomic function of the previously executed action has failed the composition core will simply drop the execution of the application skeleton and raise an error event (Activity 5). The list of actions is executed in a non-blocking fashion when we reach Activity 9. However, before that what needs to be checked is whether or not the input parameters to the atomic function need to be casted to a different type than the one expected or not before we get to invoke the corresponding functions. This is done in Activities 6,7 and 8.

5.3.1.2 Composition Core API

This section describes a preliminary API for communication between Execution Agent and Composition Core nodes. This interface provides functions for performing these tasks:

- Informing the composition core about the set of services available on a given service container (e.g. SIP container or Web Services container) which could participate in compositions.
- Consulting composition core for getting the information about next steps to be executed by the execution agents

Listing 5.7 contains the definition of the Composition Core API as Java interface. Without loss of generality, other alternatives can also be used to describe this interface such as Web Services interface specified by means of the corresponding WSDL description will be used.

Composition Core API

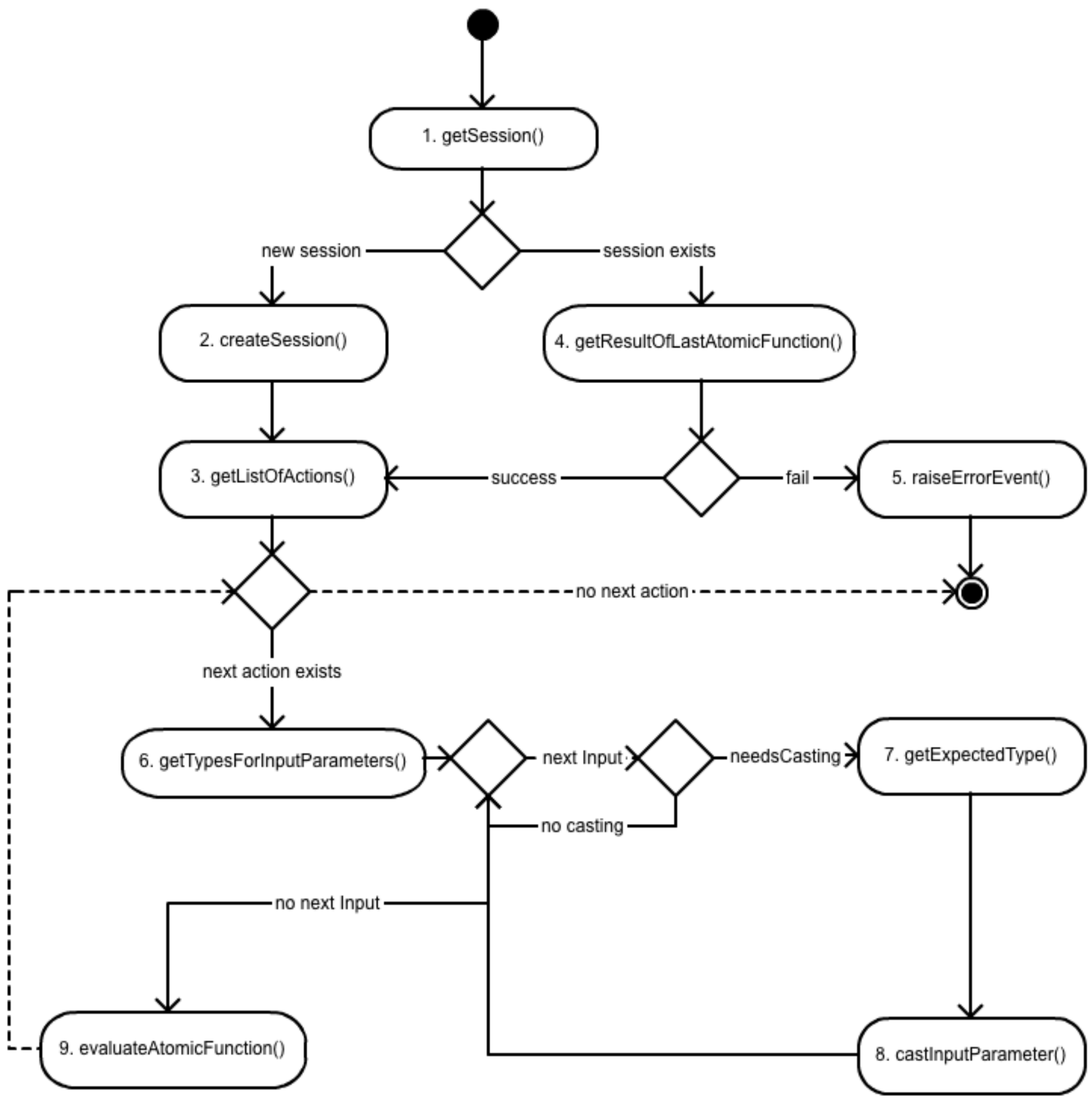

Figure 5.19: Composition core activity diagram

```
/**
 * Interface representing the API of a composition engine
 * node and  used by composition agent.
 *
 */

public interface CompositionCore {
/**
 * Provide CompositionCore with a current state of the composition
 * agent and request the sequence of services to be executed next
 *
 * @param state
 *              current state of this composition agent
 * @return sequence of services to execute next
 */

public ServiceSequence getServiceSequence(CAState state);

/**
 * Inform Composition Core about the set of (SIP) services available
    on
 * the application server and controlled by a given Composition Agent.
 *
 * @param caId
 *              id of a composition object
 * @param services
 *              set of services
 */

public void reportAvailableServices(CAId caId, Services services);

/**
 * Provide Composition Core with the current state of the composition
 * agent and request the sequence of services satisfying the
    constraints.
 *
 * This function is usually used by Web Services and other
 * API-based
 * components of the composition.
 *

 * @param state
 *              current state of this composition agent
 * @param constraints
 *              constraints for required services
 * @return
 */

public ServiceRefs getMatchingServices(CAState state,
    ServiceConstraints constraints);

}
```

Listing 5.7: Composition Core API

5.4 Prototype architecture

As described in 4.4.2, one of the main constructs of the proposed language is an action. Actions can only have local data and the only way for an action to communicate data with its environment is via effects – special variables whose value is copied to the parent, or to the adjacent action after the completion of the host action. The same notion is re-used in the way the different components of the composition framework are communicating. More specifically, all of the aforementioned components exist in isolation, having no shared state what so ever and communicate with each other by exchanging asynchronous messages and by using callbacks. The philosophy behind this interaction has been popularized by programming languages such as Erlang [10] where the developer is forced to consider a piece of software as a set of independent components that exchange messages. Moreover, when a particular component crashes, it is good practice to let the component crash and immediately spawn a new instance of that component to try and resume instead of bloating the implementation of a component with several defensive programming [96] constructs that could only take care of most common kinds of failures (let-it-crash).

5.5 Quality of source code base

There are two reasons behind this endeavor for quantifying certain aspects of our source code base. The first one is that of maintainability; Operating under the assumption that the recipients of this particular work is a group of developers that are going to be responsible for maintaining this source code base, this section aspires to become an aid for this group that helps identifying the most crucial parts of this source code base. The second reason is to try and identify the impact of applying the CPS transformation to the source code base. The following metrics have been chosen for this purpose; software lines of code, cyclomatic complexity, first order density, propagation cost and core size. Each metric is further explained in the corresponding subsections of this chapter. The approach used in this section, for quantifying the health of our source code base is similar to the one used by Baldwin et al. [12] to review popular open-source and closed source projects such as Gnucash [114], Abiword [173], Google Chrome [134], Linux (kernel) [101], Mysql [116] and others.

Recently, the same set of metrics has been used by Almossawi in an analysis of the Firefox codebase maintainability [5].

5.5.1 Software lines of code

Software lines of code (SLOC), measures the number of executable lines of code, that is by excluding blank lines and lines that contain comments. It is one of the oldest and most widely used baselines to measure software quality [2]. Intuitively, a source code base that has more lines of code is more difficult to maintain. For the purposes of this analysis, we measured the lines of codes of all java files (*. java) that were written for each of the different releases that we have made during the development of this prototype. This SLOC analysis also took into consideration JUnit [33] test files that we developed in order to assert proper functionality.

5.5.2 Cyclomatic complexity and essential cyclomatic complexity

Cyclomatic complexity has been developer by Thomas McCabe in 1976. This metric directly measures the number of linearly independent paths through a program's source code. For the purposes of our analysis, we employ a variant of cyclomatic complexity that is known as essential cyclomatic complexity. Essential cyclomatic complexity is the Cyclomatic complexity after iteratively replacing all well structured control structures with a single statement. Structures such as if-then-else and while loops are considered well structured. By removing all the structured sub graphs from the control graph and then calculating its cyclomatic complexity we calculate the essential cyclomatic complexity. A graph that has only the regular single entry/single exit loops or branches will be reducible to a graph with complexity of one. Any branches into or out of a loop or decision will make the graph non-reducible and will have essential cyclomatic complexity >2.

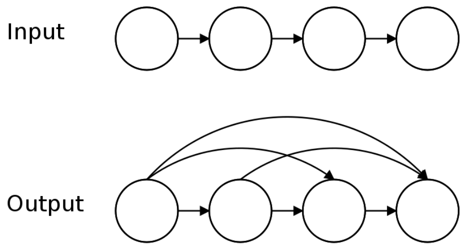

Figure 5.20: Transitive closure

5.5.3 First order density

First order density is used in order to measure the number of direct dependencies between files. It is calculated by building an adjacency matrix using a set of source code files sorted by their hierarchical directory structure. The adjacency matrix (also known as Design Structure Matrix) is a set of rows and columns. If a file depends on another file, then the intersection of those files, represented in rows and columns is marked with a number. First order density is calculated by measuring the density of the adjacency matrix. This can be done by dividing the number of non-zero elements by the product of the size of the adjacency matrix.

$$\text{first order density} = \frac{nnz(A)}{prod(size(A))} \tag{5.1}$$

5.5.4 Propagation cost

This metric is used for measuring direct and indirect dependencies between files. In practical terms it gives a sense of proportion of files that may be impacted, on average, when a change is made to a randomly chosen file. Propagation cost is calculated by calculating the matrix density of the visibility matrix that is produced from the adjacency matrix mentioned for the purposes of measuring first order density. The adjacency matrix produces the visibility matrix through transitive closure [172]. Pictorially, the function of transitive closure is shown in Figure 5.20.

The visibility matrix can be calculated by raising the first-order density matrix in consecutive powers towards reaching its transitive closure (5.2).

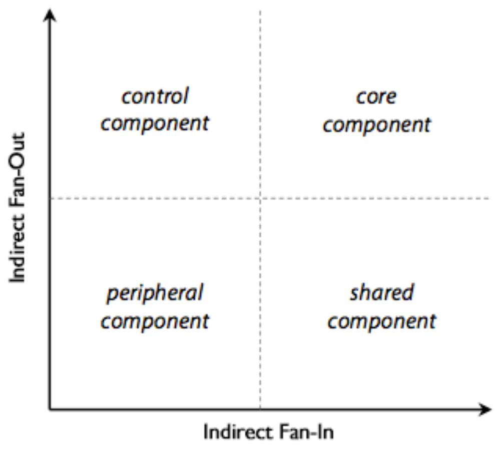

FIGURE 5.21: Fan-in/Fan-out

$$\sum_{i=0}^{N} A^i \tag{5.2}$$

However, this approach only yields specific n-level indirect relationships, for example raising the first-order-matrix to the power of two provides the second level of indirect dependencies. In our case, we chose to use the generalized approach towards transitive closure as defined by the Floyd-Warshal [126] algorithm.

5.5.5 Core, periphery, control and shared size

By creating a scatter plot of the different files available in our source code base in the axes of fan-in and fan-out we produce these four metrics. Fan-in is calculated by summing the elements of the i-th column of the visibility matrix, while fan-out is calculated by summing the elements of i-th row of the visibility matrix. In other words fan-in describes the number of components that directly or indirectly depend on i while fan-out describes the number of components i directly or indirectly depends on.

A scatter plot represents the information split in four quadrants as shown in Figure 5.21.

Core files are the files that are highly interconnected via a chain of cyclic dependencies. As such, various studies [102] have shown that core files are usually predisposed for defects. Peripheral files are the files that do not depend on a lot of other files and do not have a lot of files depend on them. Shared files are the files that do not depend on a lot of files but have a lot of files depend on them. Finally, control files are the files that depend on a lot of files but do not have a lot of files being dependent on them.

5.5.6 Data set

During the development of this prototype we used a subversion repository [38] to store the different development iterations. Overall a set of 3998 revisions has been created. From that set we promoted 11 revisions to become the releases for this prototype since each one of those represents a milestone. Those releases and their corresponding characteristics of each release are the following:

1. Coarse-grained: The original coarse-grained implementation — created for the purposes of simulating rival execution frameworks
2. Coarse-grained with HTTP EA: The coarse-grained with an HTTP EA
3. Coarse-grained with HTTP and SIP EA: Coarse grained with HTTP and SIP EA
4. Coarse-grained with AOP and events: An updated version of the coarse-grained core with support for Aspect Oriented programming constructs and events
5. Coarse-grained with HTTP EA: The updated coarse-grained core with an updated HTTP EA
6. Coarse-grained with HTTP and SIP: The updated coarse-grained core with improved HTTP and SIP EA
7. Coarse-grained with HTTP, SIP and JBI EA: Updated coarse grained core with improved HTTP, SIP and JBI EA
8. Fine-grained: The fine-grained execution framework
9. Fine-grained with HTTP EA: Fine-grained execution framework with improved HTTP EA

	Mean	Median	Standard deviation	Min	Max
Lines of code	58836.55	51193	30706.43	22210	105116
Sum essential cyclomatic complexity	6020.91	5315	2968.10	2509	10441
First order density	0.0033	0.0034	0.0003	0.0029	0.0036
Propagation cost	0.0516	0.0532	0.0150	0.0264	0.0753
Core size	0.1308	0.1703	0.0889	0	0.2164
Files	686.73	566	355.92	292	1217

Table 5.9: Cumulative statistics (n=11)

10. Fine-grained with HTTP, SIP EA: Fine grained execution framework with HTTP and SIP EA

11. Fine-grained with HTTP, SIP and JBI EA: Fine-grained execution framework with HTTP, SIP and JBI EA.

Table 5.9 shows some descriptive statistics for the 11 source code bases.

5.5.7 Data processing

To process these 11 data sets we used a program called Understand [147] in its 15-day evaluation period. Understand is a static analysis tool that can analyze .java files without the need of having those files compiled, or the requirement to have all these files collected in the workspace of an IDE, like other Java static analysis tools for Java would require. More specifically, Understand provides for us the measurements for the amount of files available in each source code base, the sum of the essential cyclomatic complexity and the amount of lines of code. In addition, understand provided for us the corresponding file dependency matrix for each source code base.

Later on we continue by processing each file dependency matrix with Mathworks' Matlab in order to produce first order density, propagation cost, core size, periphery size, control size and shared size. The basis for these six metrics is the file dependency matrix produced by Understand. The file dependency matrix allows us to create the first order density matrix, by replacing the amount of references between files with the number 1, instead of the actual count that was produced by Understand.

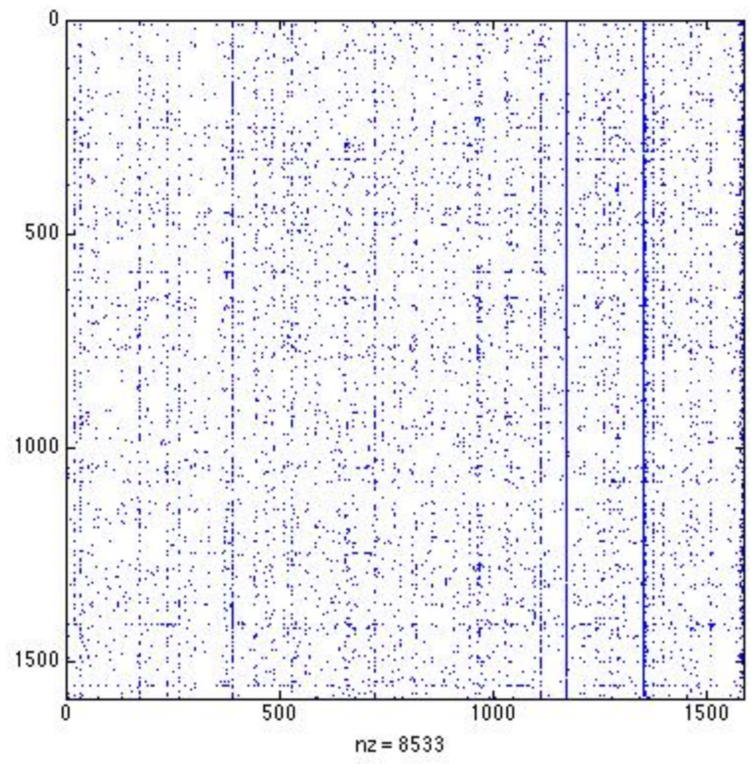

FIGURE 5.22: First order density

First-order density is calculated by computing the density of that matrix. Using transitive closure, we later on produce the visibility matrix. As shown previously in Figure 5.20 the visibility matrix describes both direct and indirect relationships among files.

An interesting observation to make here are the emerging dependencies that are produced as soon as the transitive closure is calculated (Figure 5.22 and Figure 5.23).

To produce the metrics for core, periphery, control and shared size we calculated the corresponding fan-in and fan-out of each file and later on measured the amount of each files that appears in each quadrant.

Algorithm 1 can be used for measuring the amount of file occurrences in each quadrant.

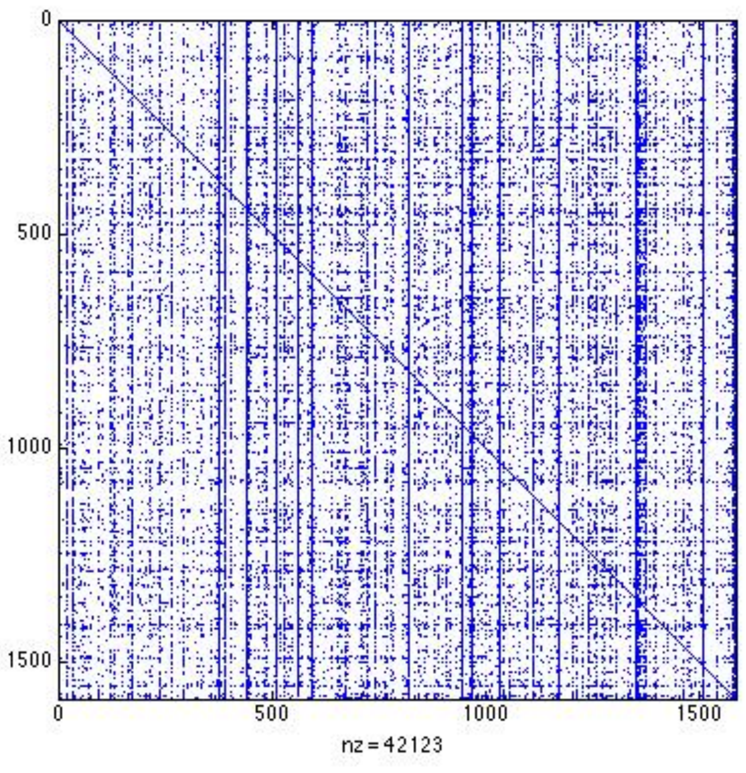

FIGURE 5.23: Visibility matrix

Algorithm 1 Counting occurrences per quadrant of fan-in/fan-out chart

```
Set A to be a two-dimensional matrix containing all elements
Set $x - boundary$ to be the middle of the x axis for matrix A
Set $y - boundary$ to be the middle of the y axis for matrix A
for $j \leftarrow 1, n$ do
    if $A(1,j) >= x - boundary$ & $A(2,j) < y - boundary$ then
        shared++
    else if $A(1,j) < x - boundary$ & $A(2,j) < y - boundary$ then
        peripheral++
    else if $A(1,j) < x - boundary$ & $A(2,j) >= y - boundary$ then
        control++
    else if $A(1,j) >= x - boundary$ & $A(2,j) >= y - boundary$ then
        core++
    end if
end for
```

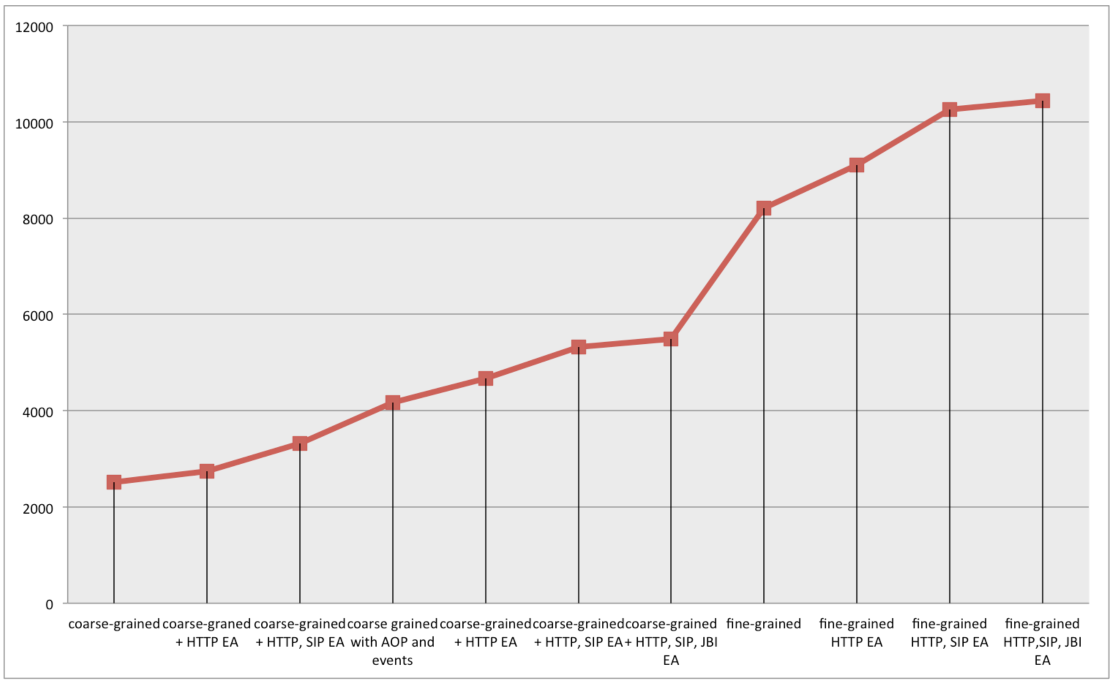

FIGURE 5.24: Sum of essential cyclomatic complexity

5.5.8 Overall findings

The sum of essential cyclomatic complexity (Figure 5.24) shows an increasing trend, especially at the point where the fine-grained version of the execution framework is introduced, where the sum of essential cyclomatic complexity jumps from 4175 to 8202 (96.45% increase). However, this increase is justified since the source code base that contains the fine-grained core also contains the coarse-grained and therefore the sum of essential cyclomatic complexity shows a significant growth. As we later on show once we describe the sum of essential cyclomatic complexity per module, the sum of essential cyclomatic complexity actually drops when the fine-grained core is analyzed in separation from the coarse-grained.

The amount of lines of code (Figure 5.25) increases in a similar pattern as the sum of essential cyclomatic complexity. Similarly to the previous figure, we notice a sharp increase in the amount of lines of code when moving from the coarse-grained core to the fine-grained one, rising from 39610 to 80335 (102% increase). As before, the increase is explained by the fact that both the coarse-grained and the fine-grained core are now included into the same source code base. Even though the choice of having both versions in the same source code base, seems redundant, functionally-wise, since the fine-grained core can imitate very nicely the characteristics of the

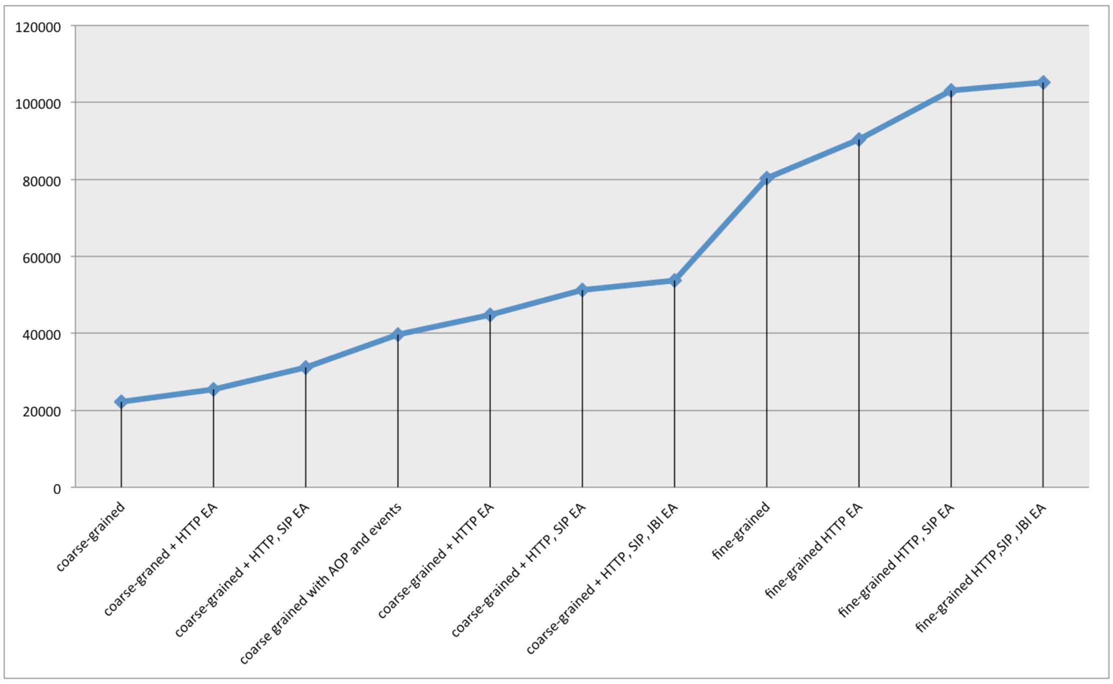

FIGURE 5.25: Lines of code

coarse-grained, we decided to keep the original core in our source code base in order to have more accurate results when comparing the performance between our proposed fine-grained core and the coarse-grained approaches used by our rivals.

Overall the observation that cyclomatic complexity increases in a similar pattern as lines of code denotes a negative attribute to the maintainability of the source code. However, in this case, the increase is justified since we are only examining a prototype that has been developed to include new features, therefore it should increase in complexity.

First order density (Figure 5.26) shows overall a decreasing trend. More specifically, the decrease is observed with the introduction of the fine-grained core. More specifically, first order density drops from 0.0033 to 0.0031 (6%) and if we compare to the first order of density of the first release the decrease is about 13%. The drop in first order density shows that the direct dependencies between files are decreasing. On average, first order density is 0.0032 with a standard deviation of 0.0002. This entails that changing any randomly chosen file has a direct impact on other 2 files.

Propagation cost (Figure 5.27) shows to increase steadily (approximately 45%), throughout the progression of the development process where the different execution agents and variations of cores are being introduced. On average we estimate the

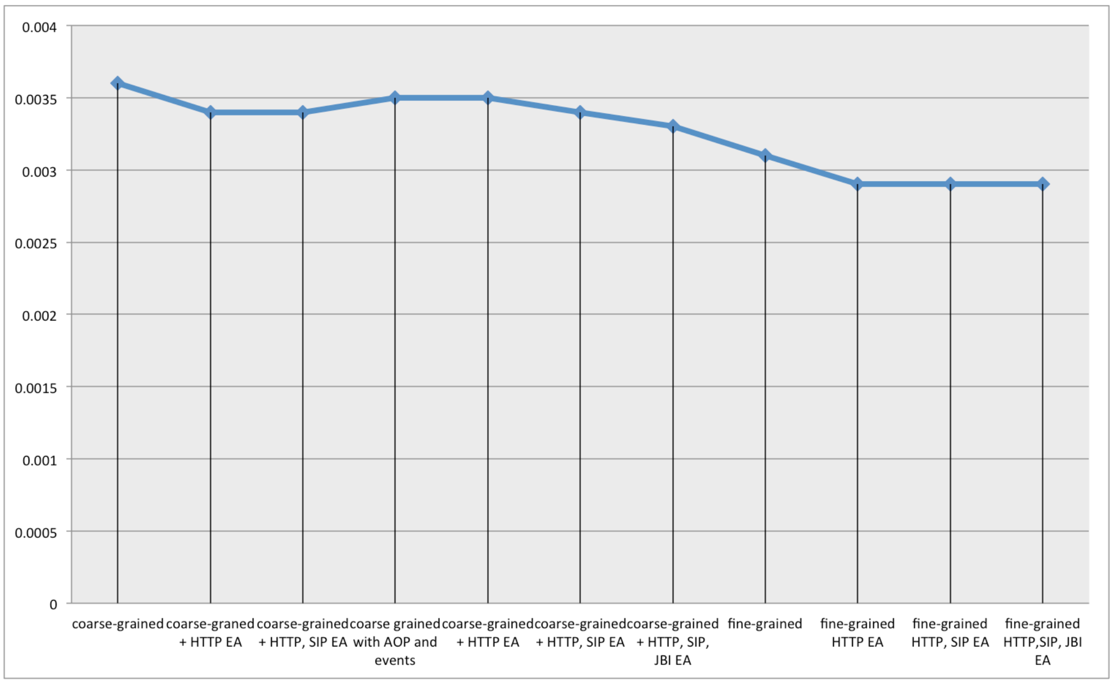

FIGURE 5.26: First order density

propagation cost to be 0.0516 with a standard deviation of 0.014. This observation entails that a change in any random file may have eventually an indirect impact on 35 other files.

Core files (Figure 5.28) are detected with the introduction of AOP and events in the coarse-grained approach. Prior to that our source code base was deprived of core files; it consisted of 96% periphery files, 1.4% control files and 2.2% shared files. Thereafter, we notice that core-size decreases with the introduction of the fine-grained approach, only to increase later on to the same degree as it was with AOP event based, after the introduction of the HTTP, SIP and JBI execution agents.

5.5.9 Per module findings

On a per module level, we proceed to analyze the different key components of our source code base in isolation, as opposed to the previous measurements which in some cases included combinations of components. In this section, the fine-grained core, the coarse grained and the HTTP, SIP and JBI EAs are decoupled and are analyzed separately.

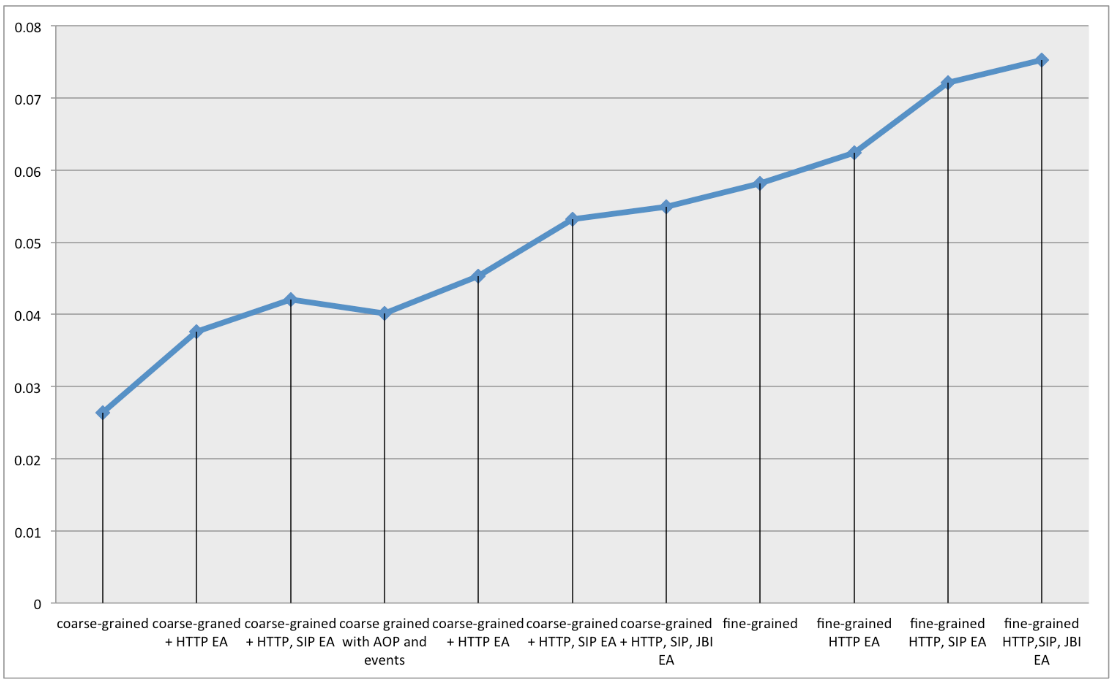

FIGURE 5.27: Propagation cost

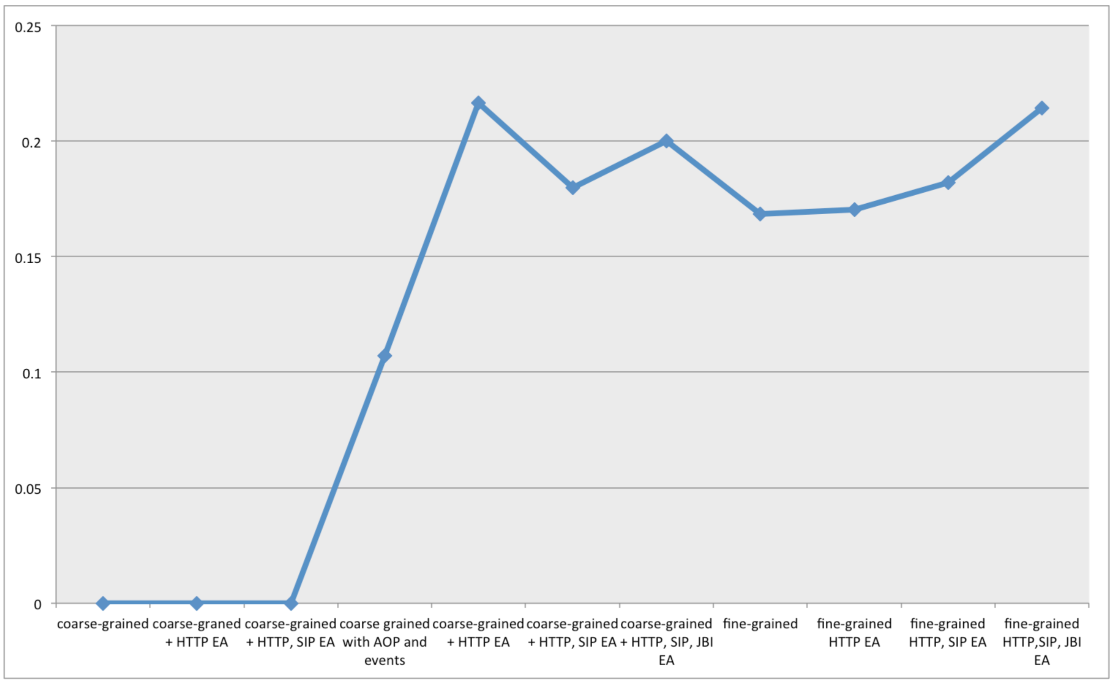

FIGURE 5.28: Core size

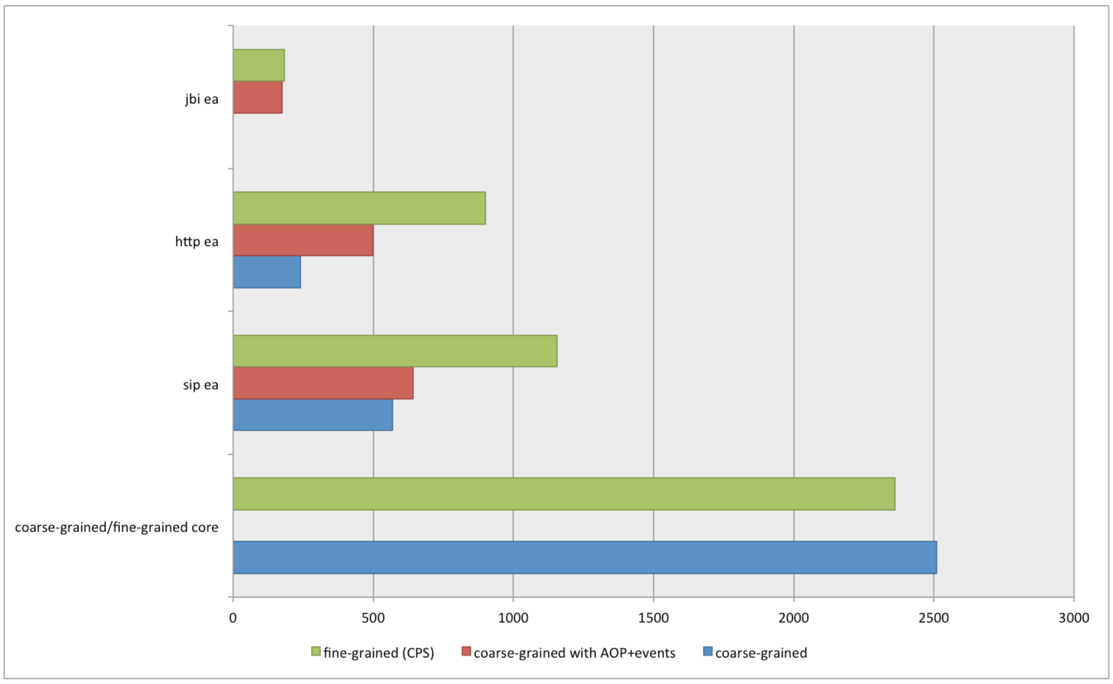

FIGURE 5.29: Essential cyclomatic complexities per module

The main motivation behind undergoing a per module analysis of our source code base was to have a clearer understanding of the impact, the CPS transformation had in our source code base, in the process of transcending from a coarse-grained to a fine-grained core. To our surprise, we observe (Figure 5.29) that the complicated process we underwent, in order to migrate from a coarse-grained composition core, to a fine-grained one, has yielded in a reduction in the sum of cyclomatic complexity. Even though the reduction is small, from 2509 to 2361 (5% decrease) it is an important one since the fine-grained core is capable of performing the same functionality as the coarse-grained one (blocking calls in synchronous mode) and its expressed with a smaller amount of essential cyclomatic complexity. However, the same trait is not maintained in the execution agents. The transition from a coarse-grained to a fine-grained core yields increases in the sum of essential cyclomatic complexity for the SIP and HTTP EAs. More specifically for the SIP EA, we see two consecutive increases of 13% and then 80%. The HTTP EA shows similar traits, receiving increases in complexity of 107% and 80%. JBI Execution Agent (JBI EA) due to its simplicity only sees a 5.1% increase. The increase observed in the EAs is well justified since the fine-grained approach requires each EA to break its functionality in several phases that are later on reported back to the core.

With regards to first order density (Figure 5.30) we see that the fine-grained core

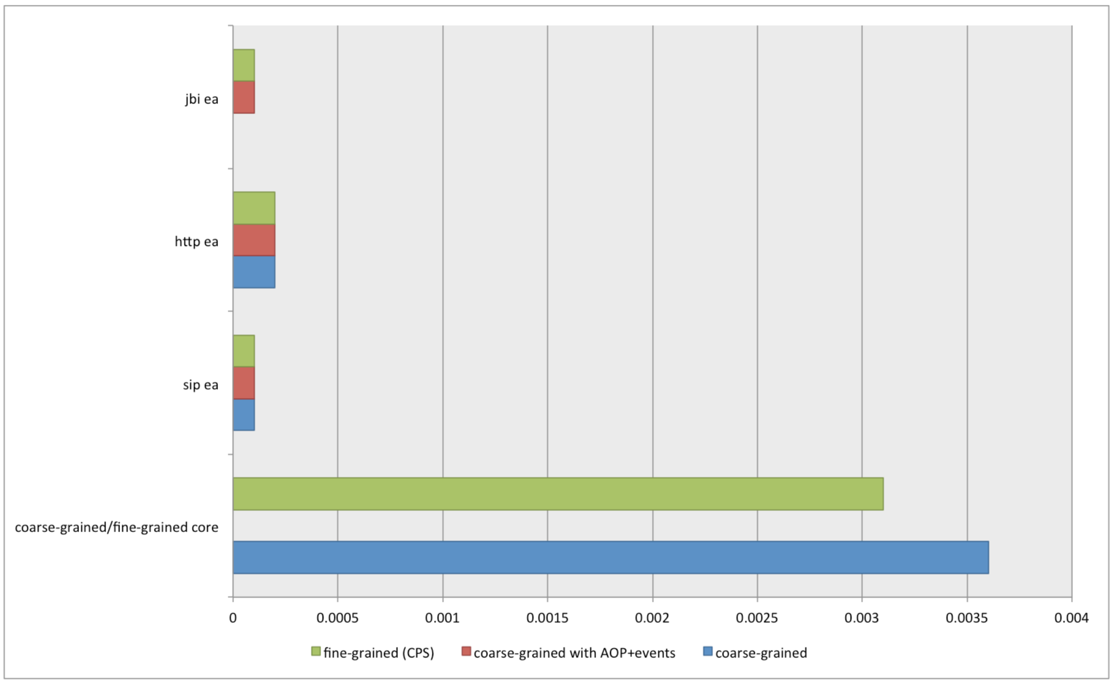

FIGURE 5.30: First order density per module

has a smaller first order density that the coarse-grained one, decreasing from 0.0036 to 0.0031 (13%). First order density remains stable for the SIP, HTTP and JBI EAs.

We notice a sharp increase in the propagation cost (Figure 5.31) for the fine-grained composition core, increasing from 0.0264 to 0.0582 (120%). Each EA also sees an increase in propagation cost. SIP EA increases 0.0045 to 0.0079 (75%) and then to 0.0097 (22%). HTTP EA increases from 0.0042 to 0.0052 (23%) and then to 0.0112 (115%). JBI EA increases from 0.0017 to 0.0032 (88%). The increase in propagation cost is justified, since the updated EAs now communicate with fine-grained core using callbacks, and therefore they are indirectly dependent to a greater amount of files than they used to.

Finally yet importantly, the fine-grained core has a core size (Figure 5.32), as opposed to the coarse grained one that is deprived of one as we have mentioned earlier. The significant difference (1710%) between the core size of the fine-grained core and that of the EA is justified since the core is much larger and much more integral to the architecture as opposed to the EA. The shift to the fine-grained core also yields increases in the core sizes of the EAs. While all the EA seem to be deprived of a set of core files in the coarse-grained version, the acquire one after the introduction of

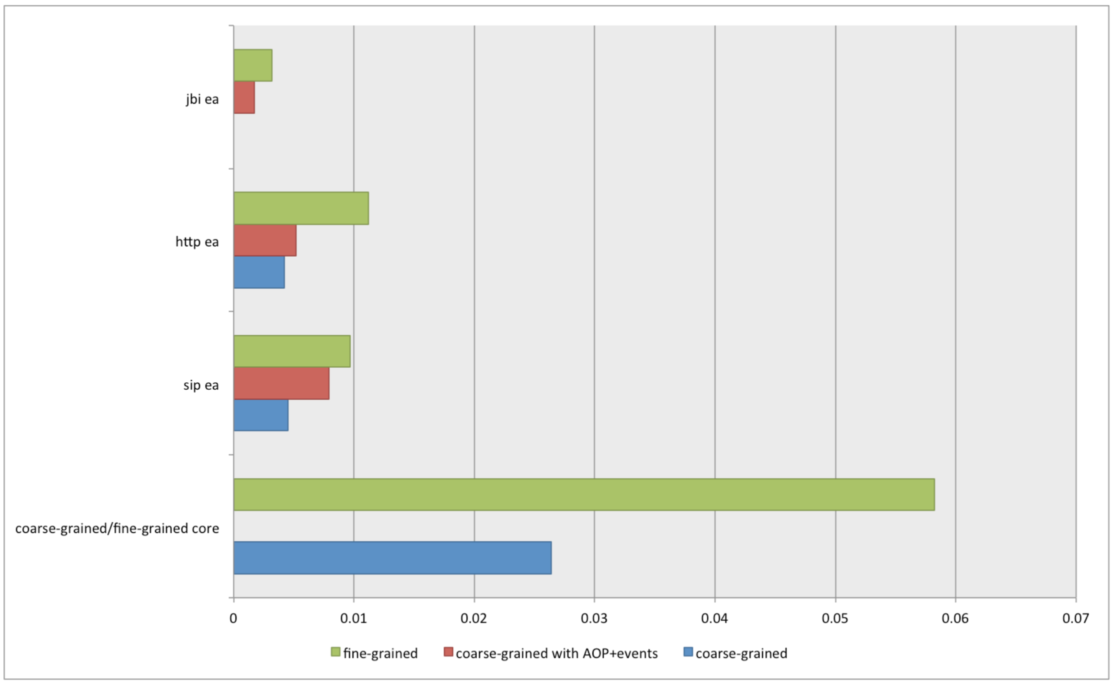

FIGURE 5.31: Propagation cost per module

Aspect Oriented Programming (AOP) and events and later on with the fine-grained core. SIP EA's core size increases from 0.0079 to 0.0097 (22%). HTTP EA's core size increases from 0.002 to 0.0093 (365%). JBI EA's core size increases from 0.0202 to 0.0323 (59%).

5.6 Summary

The outcome of this work is an asynchronous composition core that is scalable; limited by the amount of available memory and not by the number of available processing threads. Another characteristic of the asynchronous composition core is robustness as system starvation is not possible. These findings are later on evaluated empirically in 6.1. In this section we have provided implementation details that describe in detail the internal operations of the composition core, a module that can be seen as an interpreter of a concurrent programming language with a optional type system, the application skeleton repository that employs a graph database in order to deal with data dependencies, the shared state manager which is a distributed key value store where session information is maintained for every instantiation of an application skeleton, a services repository that maintains a collection of service

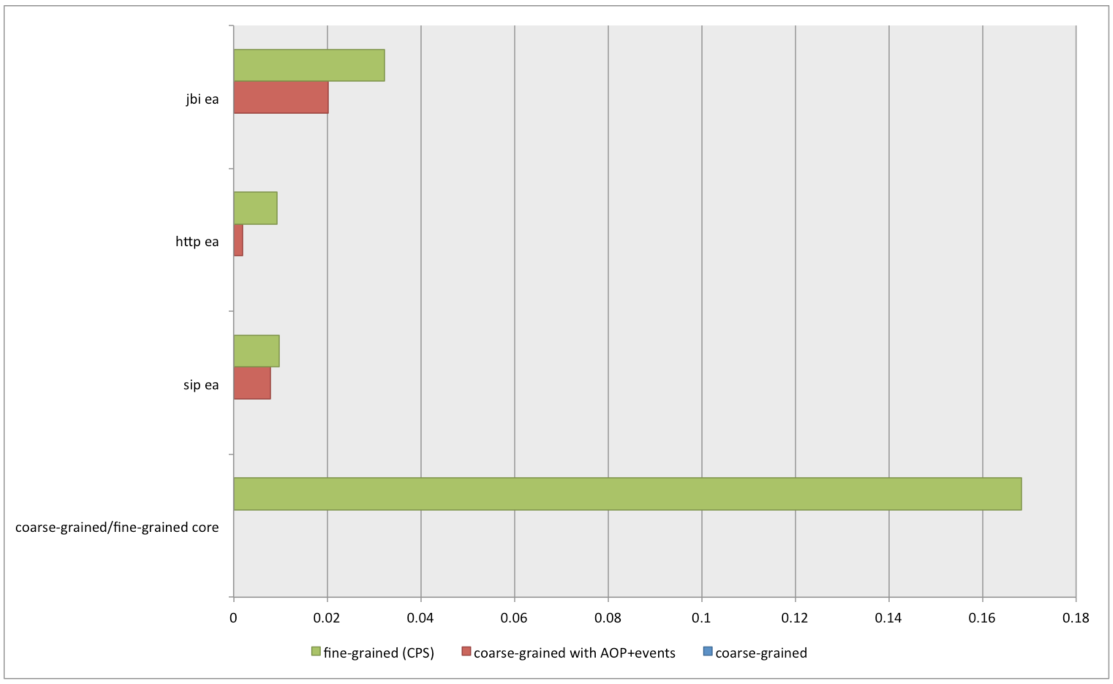

FIGURE 5.32: Core size per module

descriptions for pre-existing functionality that can be reused by a designer, the SIP execution agent that deals with SIP services, the HTTP EA, that deals with HTTP based services such as RESTful and SOAP based services. In addition, we have detailed how these components exchange messages during the lifetime of an application skeleton instance.

The resulting prototype contains 105116 lines of executable code. Our findings show that on average 13% of files are highly interconnected. From a first-order density point of view we find that changing any randomly chosen file has a direct impact on other 2 files, and ultimately it may have an indirect impact on 35 files. The application of CPS in our coarse-grained core, yields a fine-grained core that allows for non-blocking calls, but can also perform functionally in the same fashion as the coarse-grained one and at the same time is expressed programmatically with a smaller amount of essential cyclomatic complexity. However, the application of this style places a higher burden on the EAs since, even though they seem to maintain the same amount of first order density, they acquire a higher propagation cost and a higher core size.

Chapter 6

Evaluation

This chapter provides an evaluation of our work. For this to be accomplished, different aspects of the composition framework and SCaLE are evaluated using different evaluation criteria. We start by evaluating the composition framework. More specifically, we measure the overhead (delay) introduced by the composition framework and verify its robustness under heavy load. Our aim is to show through empirical experimentation that the overhead introduced by the composition framework is minimal and that the system is stable after running for a long period, having received a large amount of requests.

Then we move on to a qualitative evaluation of SCaLE. More specifically we compare SCaLE to its two closest rivals, WS-BPEL and BPMN 2.0. Even though our comparison focuses intentionally on these two languages, due to the fact that they are widely used and both have formally defined execution semantics, the qualitative evaluation approach can be used to compare our language with any other language that is available in the state of the art. The qualitative evaluation compares these languages in terms of abstraction, flexibility, support for the integration of services with different interfaces using different communication protocols, usability of the language for business process developers without an IT-background and finally support of workflow patterns.

6.1 Composition core performance analysis

We begin our evaluation by comparing variant configurations of our fine-grained composition core to a coarse-grained, dedicated thread implementation that aims at mimicking the behavior of the closest rivals to this proposed approach which are systems such as ActiveBPEL, so workflow engines that have been designed to be deployed to as servlets on top of containers. In a coarse-grained dedicated thread approach [50], dedicated threads handle requests to the composition core: each one is responsible for the execution of each action in an application skeleton until its completion. Invocations of external services, performed by service template actions, are blocking the threads until a result is available.

In our fine-grained composition core, the execution of an application skeleton is divided into smaller tasks. Thereafter, several pools of worker threads that are not tied to a specific application skeleton process these tasks. Whenever an application skeleton has to wait for the execution of an external service, the corresponding task is put on a waiting list and will be retrieved by a worker thread together with its context information when the result of the external invocation is available. Some improvements to the supported protocols, such as the support for Comet [49] in HTTP allow asynchronous notification of the caller when a result is available. Together, these various improvements allow the composition core to be fully asynchronous and to avoid blocking threads and other resources during the execution of an application skeleton.

The fine-grained core can be deployed in two ways: inside a container (usually the application server SailFin) or standalone. In the latter case, the composition framework provides its own HTTP server (set of HTTP listening threads) instead of relying on the infrastructure provided by the container. Standalone deployment reduces the overhead that is added by the application server and allows for a more direct management of threads and resources.

In order to evaluate these improvements, a number of performance tests have been done with the coarse and fine-grained versions of the composition core with two main goals:

- Measure the overhead (delay) introduced by the composition framework

- Verify the robust behavior of the composition framework under heavy load.

For most of the tests, five different variations are compared:

- The coarse-grained composition core inside SailFin
- The fine-grained composition core in ”synchronous mode” inside SailFin
- The fine-grained composition core in ”asynchronous mode” inside SailFin
- The fine-grained composition core in ”synchronous mode” without a container (standalone)
- The fine-grained composition core in ”asynchronous mode” without a container (standalone)

The following two subsections 6.1.1, 6.1.6 describe our findings when testing the different variations from the HTTP and SIP perspectives.

6.1.1 HTTP tests

This subsection describes the testing environment utilized for our HTTP tests. A test client makes a HTTP request to the composition framework (CF) and invokes an application skeleton. This application skeleton will in turn invoke a single external service over HTTP. This test service returns a response after a specified delay, in order to simulate what happens with real external services that can take an arbitrary amount of time to deliver their response. The response ends the execution of the simple application skeleton and allows the composition framework to return the result to the test client.

Some of the tests were done using different application skeletons: a special application skeleton that was returning immediately and another one that provided a delayed response without invoking any external services. This allows us to compare with and without external services and to evaluate how much the invocation of external services contributed to the end-to-end delay.

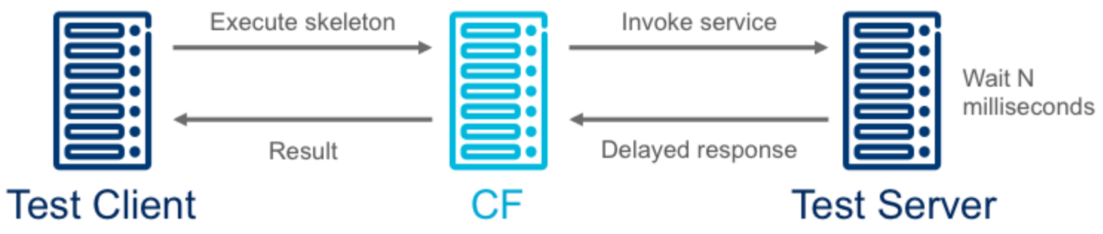

FIGURE 6.1: Test setup

The tests were run in two different configurations: one in which the test client, composition framework and test server were deployed on three different machines connected by a 100Mbit Ethernet network, and another configuration in which all three were deployed on the same machine. The first configuration minimizes the impact on CPU and memory that the client and server could have on the execution of the composition framework. The second one minimizes the impact of the network and interference from other machines that were also connected to the same switch.

In both cases, the machine running the composition framework had these characteristics:

- CPU: Intel(R) Core(TM)2 Duo CPU E6550 @ 2.33GHz
- RAM: 8 GB
- Operating system: Linux (Ubuntu 10.04, kernel 2.6.32-26-server)
- Java version: 1.6.0_20
- Java configuration: -Xms128M -Xmx256M (standalone) or -Xmx512m (SailFin)

In the ”split” configuration, the machine running the test client was identical to the one running the composition framework, except that it was running a slightly different version of Linux (Debian 5.0, kernel 2.6.26-2-amd64). The machine running the test server had these characteristics:

- CPU: Intel(R) Xeon(R) CPU E5504 @ 2.00GHz
- RAM: 8 GB
- Operating system: Linux (Debian 5.0, kernel 2.6.26-2-amd64)

- Java version: 1.6.0_22

The test client that we used for HTTP tests was JMeter [76]. We configured it to simulate a number of concurrent clients (between 100 and 500 or more) making requests to the composition framework in a loop. Most tests were run with 10000 to 30000 requests for each measurement point. The end-to-end duration of each request was measured, as well as the throughput in number of requests per second. In addition we collected information about the error rate, minimum response time, maximum response time, 90% percentile for the response time and overall duration.

The test server that we used was written by us for these performance tests. Its main function was to accept HTTP requests and wait a specified amount of time before returning. This functionality was implemented were easily by utilizing the Thread.sleep(duration) function provided by Java.

The tests were automated as much as possible so that we could run them with the five deployments of the composition framework: coarse-grained dedicated thread, fine-grained with and without SailFin and in synchronous or asynchronous modes. During the tests we varied the number of simultaneous clients, the delay on the server side, the number of listening threads for the composition framework, etc.

The following JVM options have been used [113]:

-server: a virtual machine (VM) tuned for server that is specially tuned to maximize peak operating speed — it is intended for executing long-running server applications, that need the fastest possible operating speed more than a fast start-up time or smaller runtime memory footprint

-Xms2048m: This option sets the initial minimum Java heap size

-Xmn128m: Sets the heap size for the young generation

-Xmx2048m: This option sets the maximum Java heap size

-XX:+UseParNewGC: Enables the parallel copying collector; this collector parallelizes the copying collection of multiple threads which is more efficient than the original single-thread copying collector for multi-CPU machines

-XX:+UseConcMarkSweepGC: Enables the concurrent low pause collector

-XX:ParallelGCThreads=2: Sets the number of garbage collector threads

-XX:NewRatio=2: Ratio of new/old generation sizes

-XX:+UnlockDiagnosticVMOptions: Enable processing of flags relating to field diagnostics

-XX:SurvivorRatio=8:Ratio of eden/survivor space size

-XX:PermSize=64m: Default size of permanent generation

-XX:MaxPermSize=192m: Maximum size of permanent generation

-XX:LargePageSizeInBytes=2m: Large page size to let VM choose the page size

-XX:MaxTenuringThreshold=8: Maximum value for tenuring threshold

-XX:CMSInitiatingOccupancyFraction=1: Sets the percentage of the current tenured generation size

Moreover, the composition framework was configured to exchange information among its constituent components as Plain old Java Objects (POJOs) [59] in order to minimize serialization/deserialization overhead, debug information was not collected and stored and last but not least session information per application skeleton instance was removed after the completion of each application skeleton.

After a first series of tests, we discovered that under a very high load, the composition framework seemed to perform well for the first few thousand requests, but after a while it produced a rapidly increasing number of errors. Our test server also exhibited the same behavior. We identified this problem as being caused by the default limit of file descriptors that a process can open simultaneously (in this case, TCP sockets). Increasing this default number in /etc/security/limits.conf from 1024 to 60000 solved the problem and allowed us to run the tests even with a very high number of simultaneous requests.

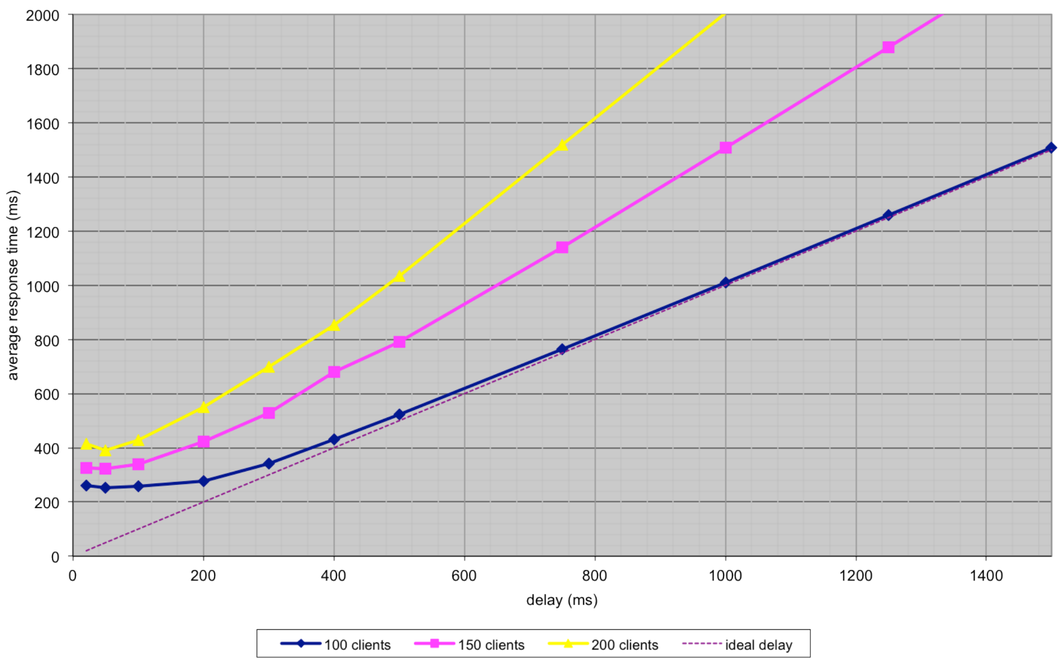

FIGURE 6.2: Average response time of coarse-grained approach in Sailfin (100 listeners)

6.1.2 Coarse-grained vs fine grained task granularity — response time

A first series of tests that immediately shows the advantages of fine-grained task granularity of the asynchronous composition core compares the end-to-end response time for increasing server-side delays with different numbers of simultaneous clients.

In graphs 6.2 and 6.3, the dotted line shows the "ideal delay", which is the delay requested from the test service. If the composition framework, the test server and the network were perfect and had no overhead, then the end-to-end response time would be equal to the requested delay.

For these tests, the application server SailFin was configured with 100 listening threads for HTTP. This is much higher than the default 5, but we saw that it was necessary to increase it because otherwise the comparison would not have been fair since SailFin would have been unable to handle a large number of simultaneous requests. The standalone version of the composition framework was configured with its default 64 listeners, which was sufficient for this test.

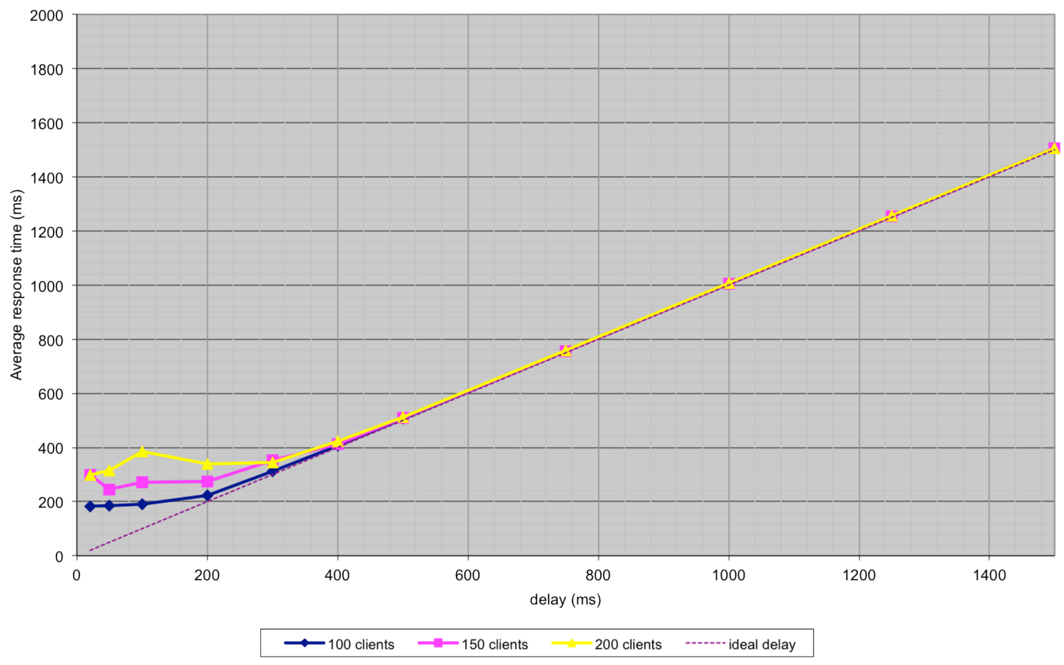

FIGURE 6.3: Average response time of fine-grained composition core

Each point on these graphs represents the end-to-end time, averaged over 30000 requests. By comparing the coarse-grained dedicated thread approach to the fine-grained composition core, we can mainly observer two significant differences:

1. The fine-grained core is always close to the ideal delay (dotted line), regardless of the number of simultaneous clients. The coarse grained dedicated thread approach performs correctly when the number of clients is equal to its number of listening threads (100) but its performance is significantly degraded when the number of client increases to 150 or 200.
2. On the left side of the graph (low delays), the fine-grained composition core performs better than the coarse-grained dedicated thread approach, which means that it introduces less overhead.

6.1.3 Coarse-grained vs fine grained task granularity — throughput

The throughput measured for the same test as above shows the same trends: the fine-grained composition core scales much better than the coarse-grained one because

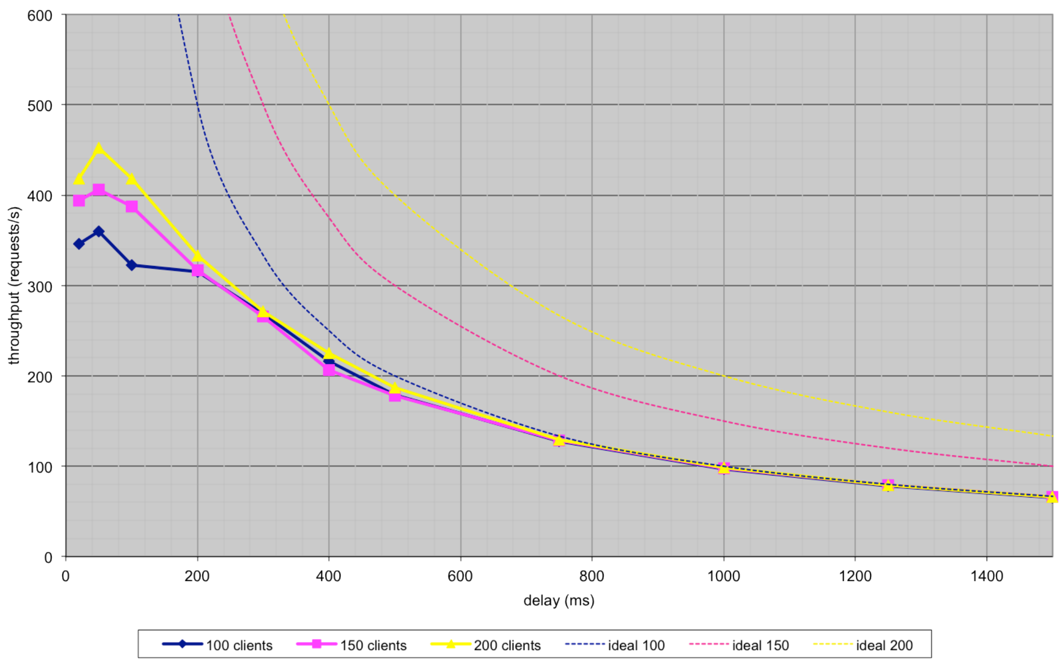

FIGURE 6.4: Throughput of coarse-grained approach in SailFin (100 listeners)

the coarse-grained dedicated thread is limited by the number of threads that were blocked while waiting for responses from the external services.

In graphs 6.4 and 6.5, the dotted curves show the ideal throughput: the number of requests per second that would be possible for 100, 150 or 200 clients if there were no other delays than the ones requested from the external server.

On the middle and right sides of the graphs, we can see that the coarse-grained dedicated thread core has quickly reached its maximum throughput and cannot serve more requests per second even if the number of clients increases. This is because all of its 100 listener threads are blocked while waiting for the responses from the external service.

In comparison, the fine-grained composition core performs better: we can see that it is close to the ideal throughput for 100, 150 or 200 simultaneous clients. The performance curves for the fine-grained composition core are close to the ideal curves. This shows that the composition framework still has some spare capacity before being limited by the CPU or network, contrary to the coarse-rained dedicated thread approach that was limited by the number of threads and the overhead that they imply. On the left side of the graphs (low delays below 200ms), we see that the coarse-grained dedicated thread approach can serve between 300 and 450 requests

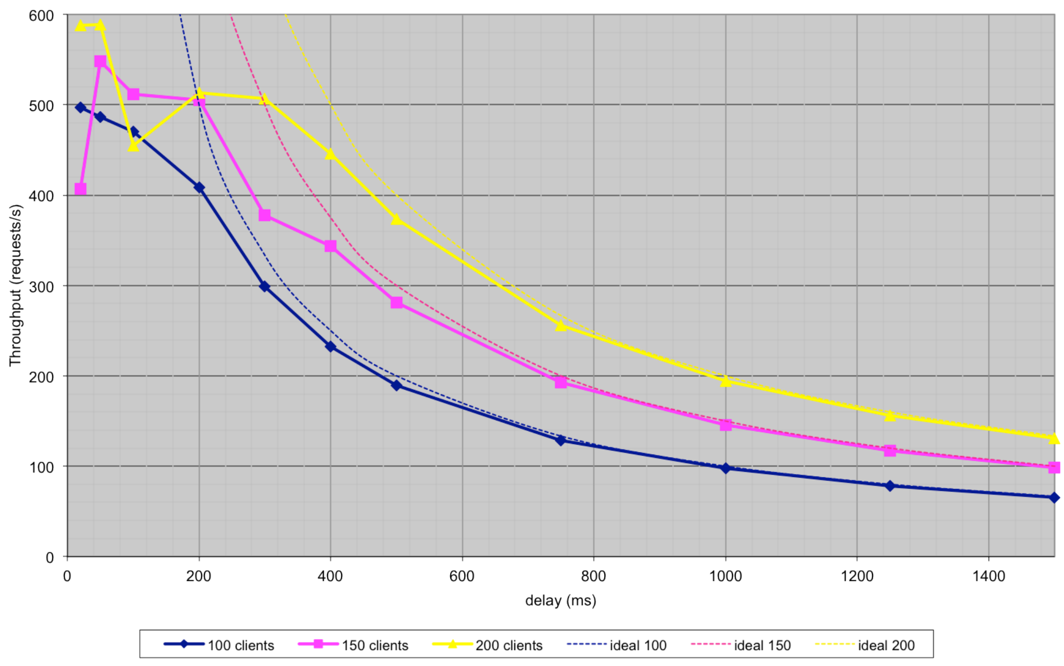

FIGURE 6.5: Throughput of fine-grained composition core

per second. The fine-grained composition core performs better and can serve between 400 and 600 requests per second.

The small dip in the yellow curve for the 100ms delay with the fine-grained composition core is most likely due to a temporary disturbance of the test machine while this test was running. A subsequent re-run of this test showed that the composition framework was able to serve a bit more than 500 requests per second instead of the 454 shown on this graph. But we decided to keep the original (unaltered) test data anyway, instead of replacing it to hide this temporary glitch.

6.1.4 Comparison with a high number of simultaneous clients

We repeated these tests with a higher number of simultaneous clients: 500 and more, up to 1000. For a very high number of clients, the coarse-grained dedicated thread approach running in SailFin started producing timeouts. Even the fine-grained composition core had some problems because the machine was running out of available port numbers: many of the TCP ports were in the TIME_WAIT state, preventing them from being reused immediately.

The graphs for 500 simultaneous clients are shown in Figure 6.6 and Figure 6.7.

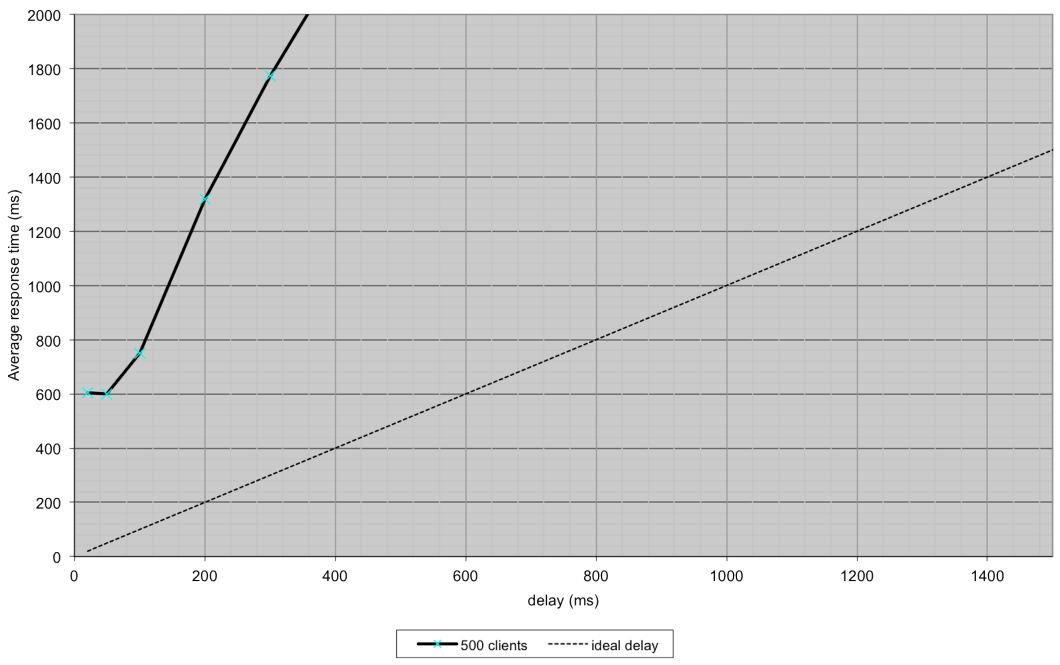

FIGURE 6.6: Average response time of coarse-grained approach in Sailfin (100 listeners) and 500 simultaneous clients

The coarse-grained dedicated thread approach could not cope with the high number of simultaneous clients because most of its threads are blocked while waiting for a response from the external service. As a result, the delays accumulate and quickly become unacceptable, causing timeouts on the client side. Although it is not visible on the graph because it is out of range, the average end-to-end response time for a server-side delay of 1500ms is around 7 seconds for the coarse-grained dedicated thread approach.

Here we see that the results of the fine-grained composition core are not as good as expected and the response times vary a lot. For relatively high delays (right side of the graph, delay >= 1000ms), the results are relatively close to the ideal case. But for the smaller delays on the left side of the graph, the average response time seems to be a bit unpredictable even if it is better than for the coarse-grained dedicated thread approach.

We identified this problem as coming probably from one of the libraries that we use for HTTP Comet handling: it does not seem to release its resources fast enough when handling a large number of requests, which causes a somewhat erratic behavior.

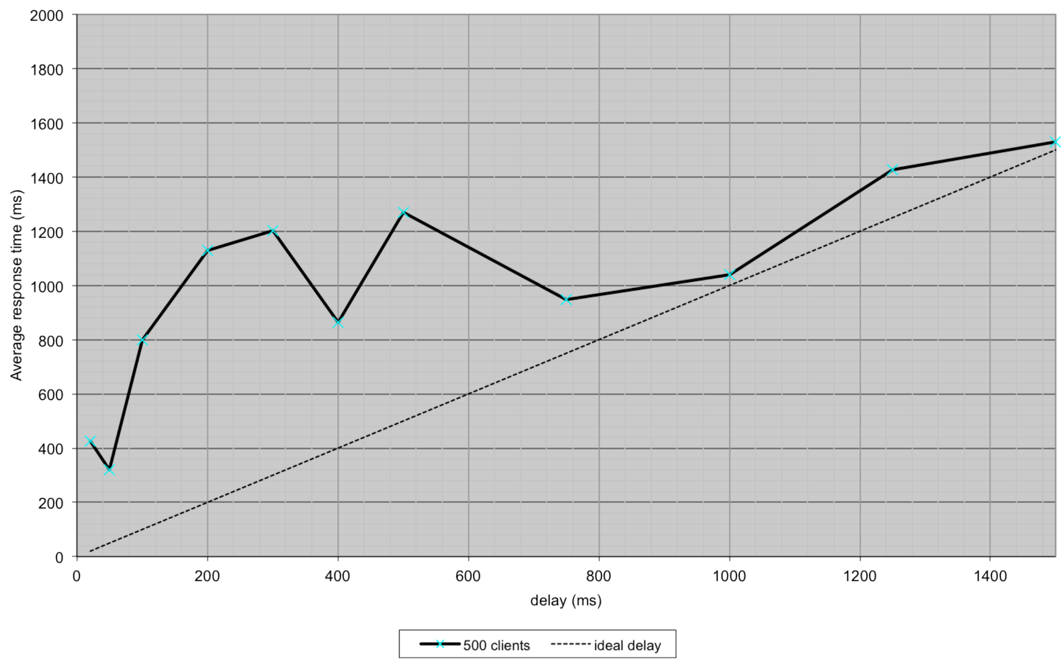

FIGURE 6.7: Average response time of fine-grained composition core with 500 clients

We are investigating this further and will try to optimize the code in order to improve the results.

6.1.5 Comparison with other configurations of the fine-grained core

The fine-grained composition core can be run in synchronous or asynchronous mode. The synchronous mode is closer to the coarse-grained dedicated thread approach although it benefits from other improvements in the code. The composition framework can also be deployed without a container (as shown in the graphs above) or within SailFin. In order to compare the performance of the various configurations and to evaluate the relative impact of the container and of the synchronous or asynchronous mode, we have also tested the following deployments of the composition framework:

- Standalone synchronous
- Asynchronous in SailFin

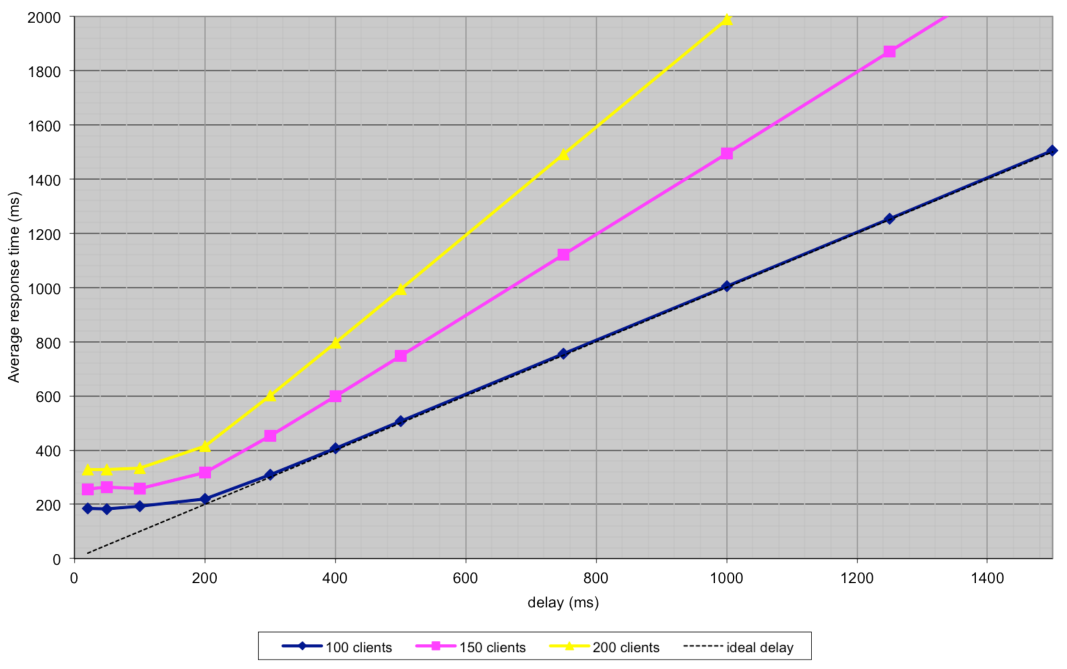

FIGURE 6.8: Average response time of fine-grained composition core in synchronous mode

6.1.5.1 Standalone synchronous — Response time and throughput

Compared to the asynchronous standalone mode, the synchronous mode shows a behavior similar to the coarse-grained dedicated thread approach when the delays are rather long: the slow responses from the external service are blocking the listening threads and cause all other requests to be queued up. The middle and right sides of the graphs above are very similar to the results of the coarse-grained dedicated thread approach.

On the left side of the graphs (delays below 200ms), the results are better than the coarse-grained dedicated thread approach but not as good as the fine-grained composition core. This means that the composition framework benefits from the improvements of the fine-grained composition core and the more efficient standalone mode, but is slightly limited by the synchronous mode.

6.1.5.2 Asynchronous in SailFin — Response time and throughput

We see in graphs 6.10 and 6.11 that the blue and purple curves (100 and 150 clients) are better than the coarse-grain dedicated thread approach in SailFin, but not

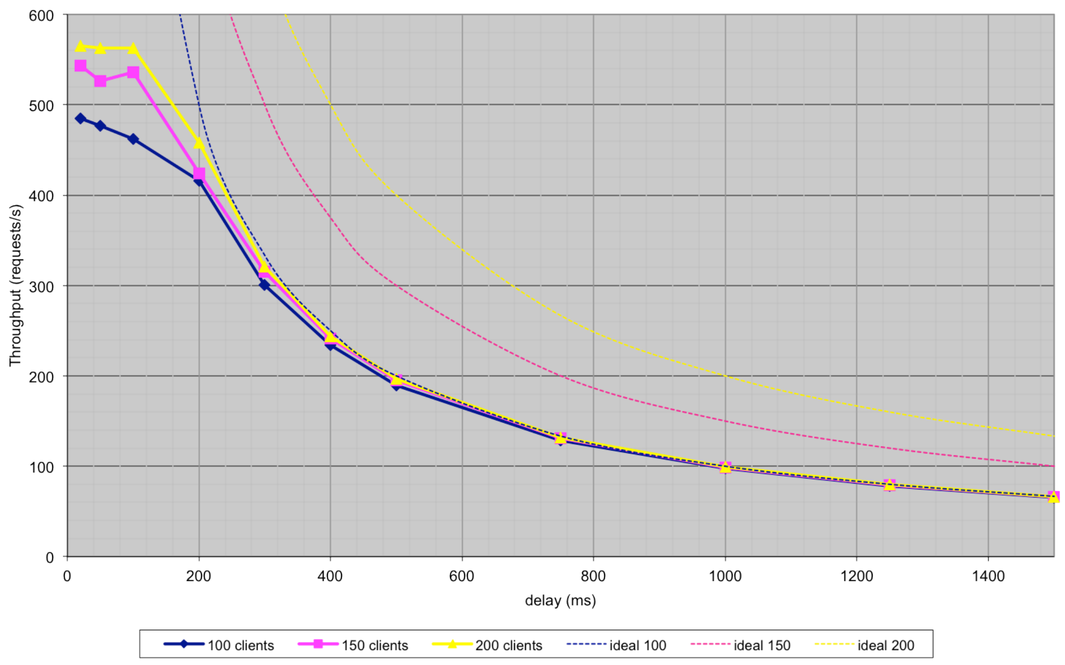

FIGURE 6.9: Throughput of fine-grained composition core in synchronous mode (100 listeners)

as good as the standalone fine-grained composition core. The difference with the standalone version for the small delays shows the overhead of SailFin.

However, we see an issue with the yellow curve (200 clients), which shows some instability for delays below one second. For longer delays it is still better than the coarse-grain dedicated thread approach, but for short delays the behavior is not as good as expected and even worse than the coarse-grain dedicated thread approach. The reasons for the rather poor performance of this configuration with 200 clients still have to be investigated. This may be due to a configuration issue or to a problem related to the interaction between SailFin and some of the libraries that we are using. But it is unlikely that the problem comes from the core of the composition framework since the same core performs well in standalone mode.

6.1.5.3 Comparison of composition framework deployments with 100 simultaneous clients

The graphs above show respectively the response time and throughput when the five deployments of the composition framework are receiving simultaneous requests from 100 clients:

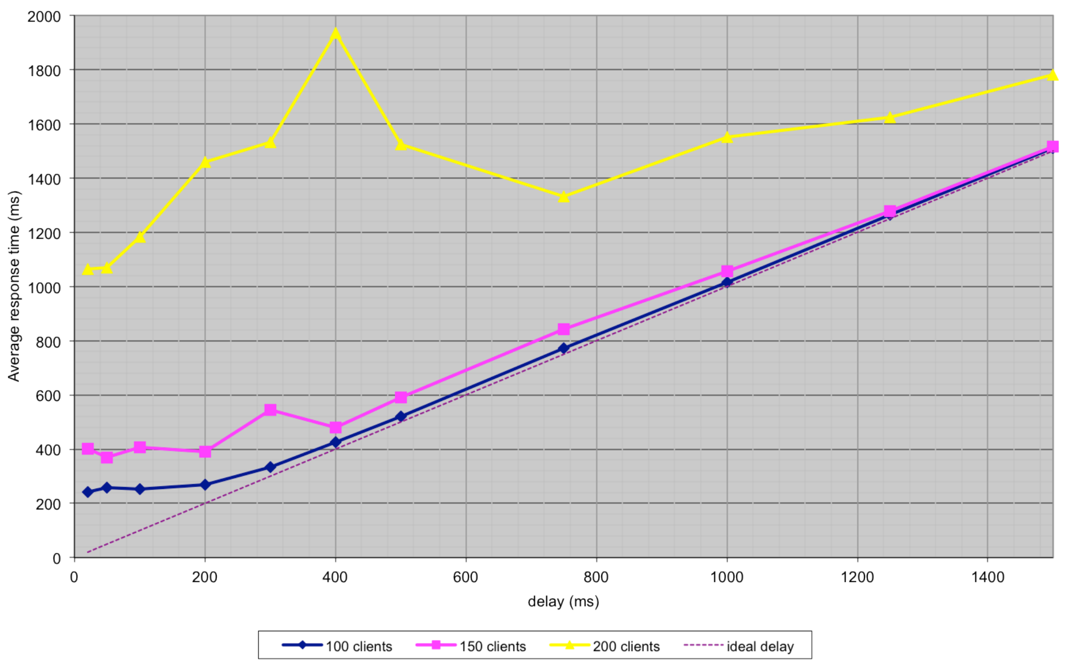

FIGURE 6.10: Average response time of fine-grained composition core in asynchronous mode in SailFin (100 listeners)

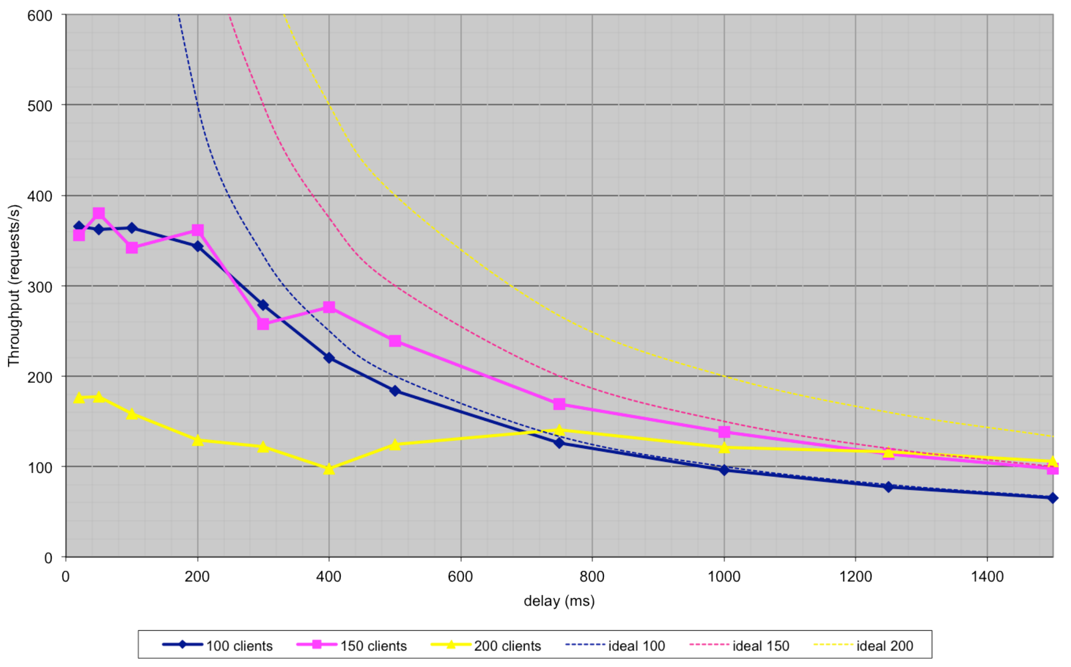

FIGURE 6.11: Throughput of fine-grained composition core in asynchronous mode in Sailfin (100 listeners)

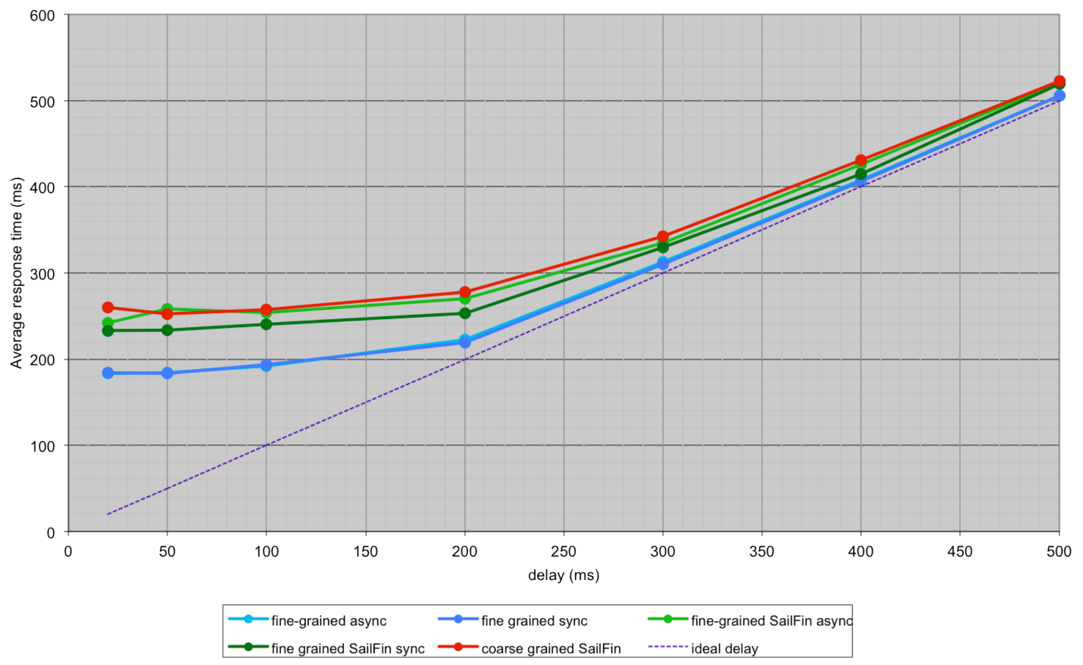

FIGURE 6.12: Average response time comparisons for 100 simultaneous clients

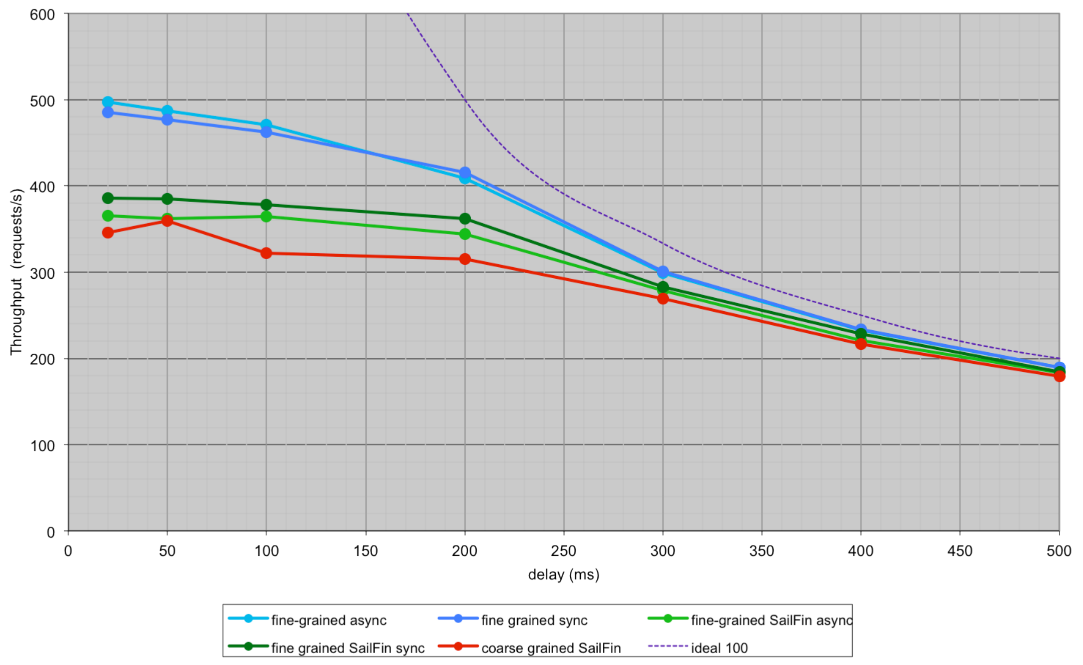

FIGURE 6.13: Throughput comparison for 100 simultaneous clients

The coarse-grained composition core inside SailFin

- The fine-grained composition core in ”synchronous mode” inside SailFin
- The fine-grained composition core in ”asynchronous mode” inside SailFin
- The fine-grained composition core in ”synchronous mode” without a container (standalone)
- The fine-grained composition core in ”asynchronous mode” without a container (standalone)

As there are also 100 listening threads, the difference between the synchronous and asynchronous modes should be minimal because each thread can take care of one client so there should be no client waiting for an available thread. This is indeed what we can see on these graphs: the asynchronous and synchronous modes of the standalone deployment (light blue and dark blue lines) are very close to each other for all requested delays, and the asynchronous and synchronous modes of the deployment in SailFin (light green and dark green lines) are also very close to each other.

We can also see on these graphs that the standalone mode is always more efficient than the deployment within SailFin: the blue lines show a better performance than the green or red lines. Among the SailFin deployments, the fine-grained composition core (green lines) is slightly better than the coarse-grained dedicated thread approach (red lines), especially concerning the throughput.

6.1.5.4 Comparison of composition framework deployments with 150 simultaneous clients

The graphs below are similar to the previous ones, except that now there are 150 simultaneous clients for only 100 listening threads. This means that in the synchronous (blocking) mode, some of the requests must be queued up while they are waiting for a thread to become available.

For the standalone deployment, we see that the light blue and dark blue lines are close to each other for low delays (100ms or less) on the left side of the graphs. But

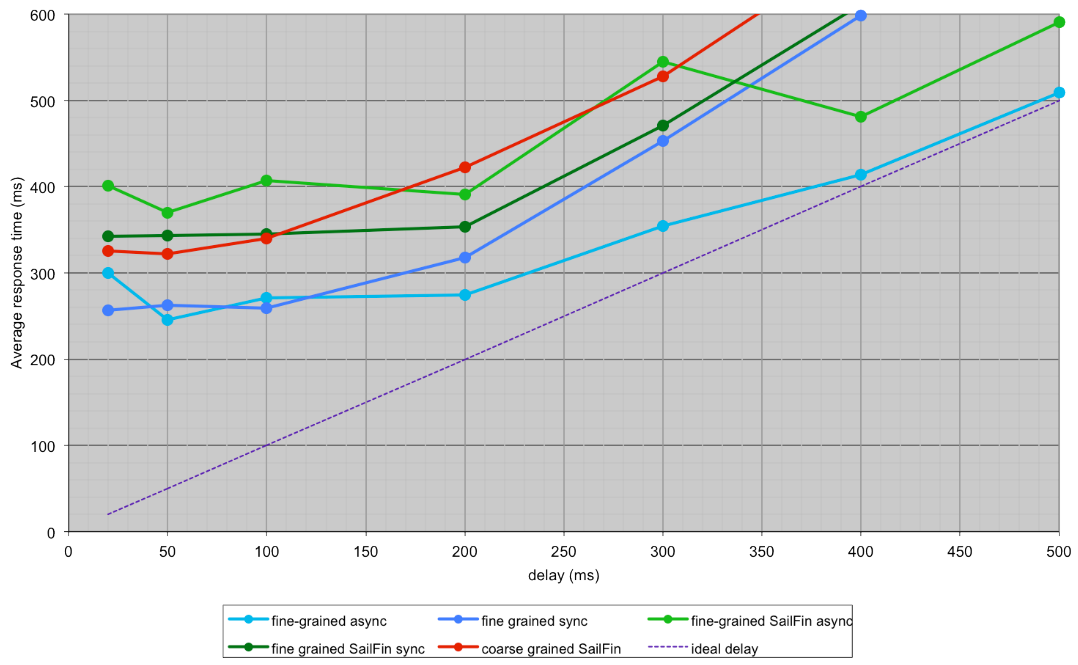

FIGURE 6.14: Response time comparison for 150 simultaneous clients

as the delay increases towards the right side of the graphs, the asynchronous mode (light blue line) gets closer to the ideal case (dotted line), while the synchronous mode adds more delays and has a lower throughput.

For the deployment inside SailFin, the behavior is the same: the light green and dark green lines get further apart from each other on the right side of the graphs.

However, we can see that the performance of the asynchronous mode inside SailFin (light green line) could probably be improved because it is not always better than the coarse-grained dedicated thread approach (red line). Again, this is probably a configuration issue or some interaction between the libraries that we are using and the container SailFin.

In summary, we have seen in this test and the previous ones that the fine-grained composition core in asynchronous mode deployed without container (standalone) always gives a better performance than the coarse-grained dedicated thread approach deployed inside SailFin, sometimes by a large margin. The fine-grained composition core deployed inside SailFin is often better than the coarse-grained dedicated thread approach, but could still be improved.

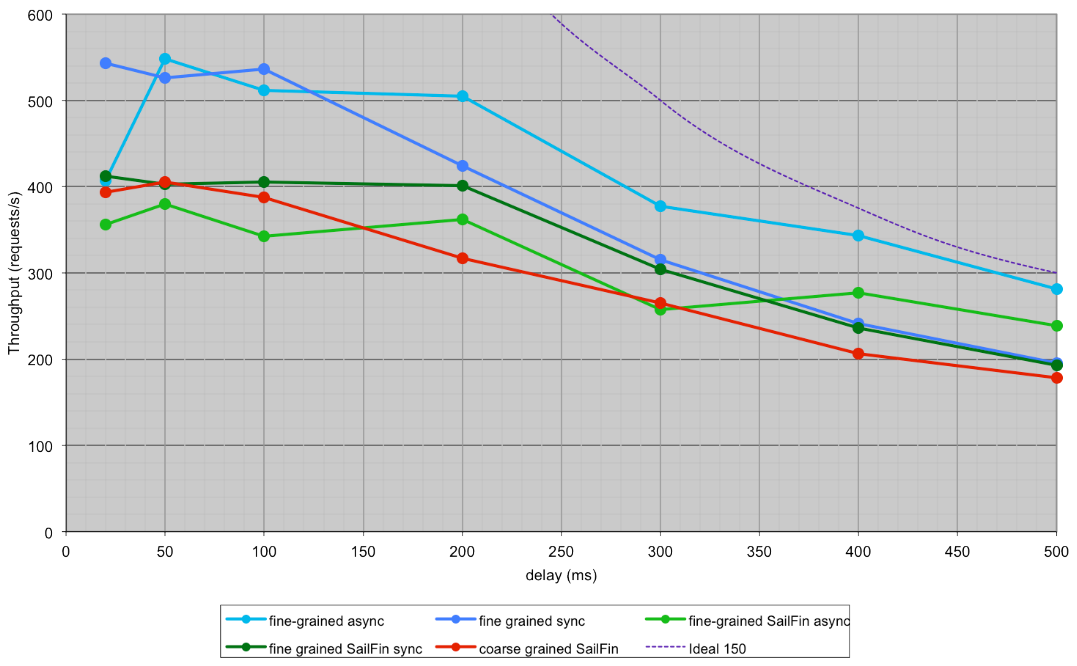

Figure 6.15: Throughput comparison for 150 simultaneous clients

6.1.6 SIP tests

Although the protocols and message exchange patterns are different for SIP and HTTP, the test setup for SIP was similar to the one used for HTTP: one SIP client, one SIP server and the composition framework (CF) between them.

For all SIP tests, the client and the server were implemented using SIPp [63] with client and server scripts derived from the standard SIPStone UAC and UAS responder. The composition framework was tested with a simple application skeleton that was triggering a simple SIP service, either as a SIP proxy or as a back-to-back user agent.

6.1.6.1 Composition framework overhead

To evaluate the performance impact of the presented solution, we run the aforementioned performance test to measure the overhead introduced by the composition framework. Our goal is to verify that the end-to-end delay added by the presence of the composition framework does not introduce significant overhead. In order to do that, we compared the performance of the system under load with and without the composition framework. The test with the composition framework involved a simple

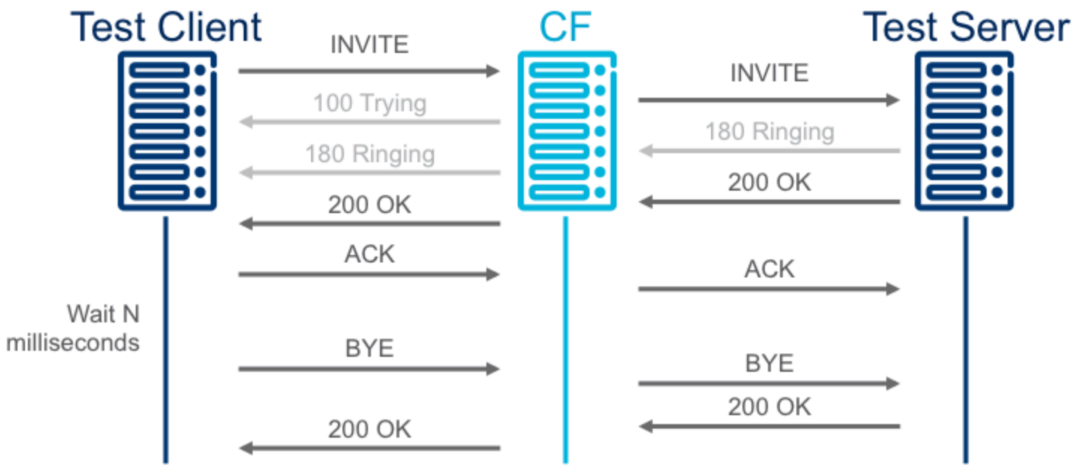

Figure 6.16: SIP test

application skeleton that inserts a single SIP servlet into the SIP chain. This servlet acts as a transparent proxy. We compared this with a similar setup in which this SIP proxy was always inserted statically, without the mediation of the composition framework.

We have run this test under medium load of 40 calls per second, which resulted in about 50% to 70% CPU utilization. In our scenario, a ”call” was defined by a sequence of seven SIP messages, illustrated in Figure 6.16: INVITE sent by the caller, three responses (100 Trying, 180 Ringing, 200 OK), an ACK, a short pause for the established call, then BYE and a response code 200 OK. Our measurements show that the average end-to-end latency without the composition framework is 5.9ms (standard deviation 2.9ms) and it increases to an average of 7.6ms (standard deviation 4.3ms) with the composition framework. The graph in Figure 6.17 shows the distribution of end-to-end latencies for two series of tests consisting of more than 20,000 calls each.

The acceptable end-to-end latency for real-time services like voice or Push-to-Talk is 1600ms [67]. This time includes the latency introduced by radio, which varies from 300ms up to 1200ms depending of technology, as well as latency introduced by other service layer nodes such as the S-CSCF. The typical requirement for an application server in the IMS context is 20ms for simple and up to 200ms for more complex applications.

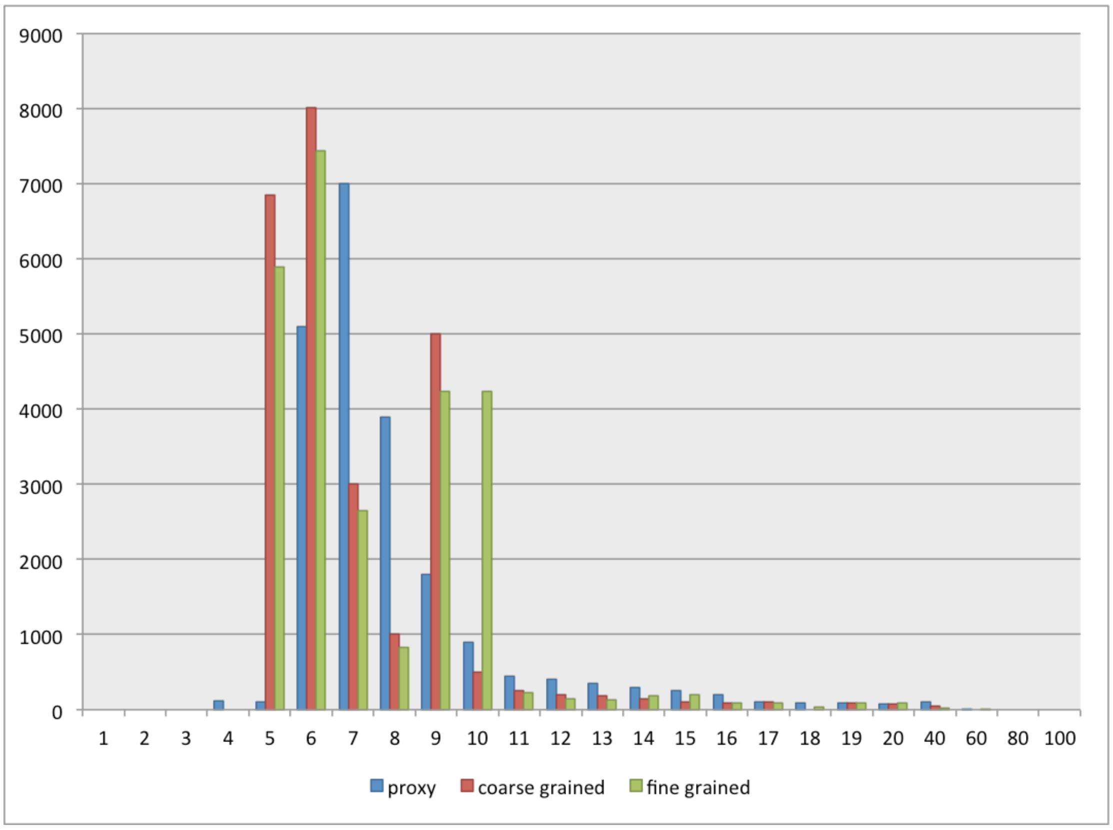

FIGURE 6.17: Distribution of call latencies

In this context, the overhead added by the composition framework is rather small: an average of less than 2ms in our test, which is less than 10 per cent of time available for simple applications or less then 0.2 per cent of end-to-end latency. The low overhead added by the composition framework makes it suitable for use in a telecommunication network, and even more with potential optimizations of software and hardware.

6.1.7 SIP and HTTP test conclusions

We measured the performance of the composition core with a large number of simultaneously running applicaiton keletons. We compared five different variations of the proposed composition core:

- The coarse-grained composition core inside SailFin
- The fine-grained composition core in ”synchronous mode” inside SailFin

- The fine-grained composition core in ”asynchronous mode” inside SailFin
- The fine-grained composition core in ”synchronous mode” without a container (standalone)
- The fine-grained composition core in ”asynchronous mode” without a container (standalone)

These tests allowed us to get absolute performance numbers for the fine-grained composition core, showing that the overhead introduced by service composition is minimal. Compared to the HTTP case, the SIP test uses three times more messages. The built-in timeouts and retransmissions that are part of the SIP protocol can cause a domino effect when the load is too high, which in turn create more load on the network and CPU, causing other requests to fail. For SIP, we measured mainly the throughput (calls per second) and the number of errors under load. It was rather difficult to get stable results because of the effects of the Java garbage collector on the system performance. From our tests we observed that the overhead introduced by the composition framework is roughly 24% compared to proxy case that does not include the composition framework.

Regarding the comparison between the different deployments of the composition framewrok, we were able to verify that our expectations were verified:

- The coarse-grained dedicated thread approach queues up requests because it is blocking during external service invocations. This can lead to system resource starvation and strong interference between skeletons when too many of them are blocked.
- The fine-grained composition core never blocks, thanks to the non-blocking invocation of external services.
- The fine-grained composition core, especially in asynchronous mode and standalone deployment, provides better throughput (requests per second) and lower end-to-end response time.

6.2 Qualitative evaluation of SCaLE

In this section we make an effort to analyze the expressiveness of SCaLE, our proposed language for service composition, by comparing it to already existing and standardized workflow languages. Even though several languages exist for the purposes of service composition and/or for the purposes of describing workflows we have focused our analysis only on those that are standardized and posses formally defined execution semantics. Those languages are WS-BPEL and BPMN 2.0.

WS-BPEL (also known as BPEL4WS) is a standardized workflow language that is widely used by the industry and academia. It has formal execution semantics but lacks a standardized graphical representation. Instead, each tool vendor has developed it's own graphical representation of WS-BPEL. That makes it hard for people using different tools to talk about process models, even if these models are essentially the same. As stated in 3.2 WS-BPEL does not implement all features required for heterogeneous service composition as described in this work. WS-BPEL could be used to implement a few of those requirements but it has not been designed with this mindset. Thus, these features can be realized in a rather complex and more verbose way in comparison to SCaLE.

The second language we took into consideration is BPMN version 2.0. We did not consider BPMN 1.0 because this version lacks a formal specification of execution semantics. In BPMN 2.0 two main features have been added in contrast to version 1.0 of the language, execution semantics and extensibility. Apart from these two features BPMN has a standardized graphical notation that makes it easier for designers with non-technical expertise to develop compositions. However, the graphical notation of BPMN uses much more symbols than SCaLE making it hard to develop error free processes. Several constructs have to be combined in order to get an executable business process using BPMN 2.0.

6.2.1 Proof-points

A number of qualitative functional requirements have been identified to formulate a set of comparative proof-points for the purposes of our comparison. This section describes these requirements, along with the point system that we introduce in order to quantify this evaluation.

Abstraction:

Abstract processes are the means to define business processes that do not explicitly specify the order in which action take place. As a hidden bonus, abstraction at this level can be used to describe a composition (or a business process) by omitting sensitive parts.

Language constructs must not be dependent on the IT infrastructure of the underlying execution framework: This requirement is important for the development of a future proof language. It should be easy to integrate new service technologies that can be used by the language constructs. Thus, business process designers should not be aware of communication protocols or interface descriptions of services that are used within a workflow. The human business process designers should be able to create business processes on a technology agnostic level.

Control flow abstraction: This requirement is related to the case handling approach introduced in [167]. Execution framework decides which actions are executed in which order: The order of actions is determined at runtime using preconditions attached to each action. Execution framework would then determine the execution order that fulfills these preconditions.

Deadlock-free: Past approaches defining workflow languages showed that the most successful workflow languages only allow to model cycle free business process models. In BPEL this is achieved by using a block structured business process models.

Flexibility:

Dynamic Service Selection (DSS): Services implementing activities of a business process are assigned at runtime using a service repository. The decision which service is assigned to implement an activity in a business process is taken based on a query being present in the business process. DSS is a means to introduce an abstraction layer between the services layer and the business process layer. Another concept, which introduces more flexibility in the execution of business processes, is late binding of services: Services are selected based on abstract descriptions at runtime.

These descriptions are along with constraints employed to select the right service at runtime.

Compensation handling is a concept needed to compensate long running processes that are faulted. With such a process it could be the case that several transaction are already committed. Thus, it is not possible anymore to role them back, when a failure in the process occurs. This is the motivation for compensation handling. A business process designer should be able to define compensation blocks within a composition that are executed in case of a failure of the main business process.

An aspect making a workflow language future proof is **extensibility**. A workflow language should provide an extension mechanism, where new functionality can be introduced without changing the core structure of the language itself.

Another concept, which introduces more flexibility in the execution of business processes, is **late binding** of services: Services are selected based on abstract descriptions at runtime. These descriptions are along with constraints employed to select the right service at runtime.

Complex message exchanges: In many cases it is useful to use message exchange patterns different from the ordinary request and reply message exchange pattern. An example for another message exchange pattern is the send and forget pattern. Using this message exchange pattern a message is sent and no acknowledgement or answer is required from the receiver of the message.

Correlation mechanism: With this mechanism it is possible to route messages arriving at a business process execution engine to the corresponding business process instances being executed in the execution engine. Correlation is a concept directly connected with asynchronous messaging.

Additional criteria:

Support for the integration of services with different interfaces using different communication protocols: This requirement comes from the telecommunications domain. Services used in this area are often built on legacy technology. There are also protocols being used, which have been developed to meet the real time responsiveness requirements telecommunication systems have. Apart from that there is a need to provide for so-called "value added" services that often use different

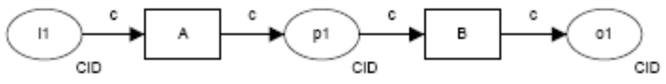

FIGURE 6.18: Sequence workflow pattern

technologies and protocols than the pure telecommunication services. Thus, it is important for the new language and the underlying execution framework to support the integration of services using different technologies and interfaces.

Usability of the language for business process developers without an IT-background: It is hard to measure the usability of the language for developers without an IT-background. Our approach to measure the usability of a workflow language is to take into account certain properties of the languages that make them more abstract or more concrete. In most cases, it is easier for people without an IT background to relate between physical and virtual representations of items, as long as the virtual representation maintains similar traits as the physical. As an example, a software calendar should look the same as a paper calendar showing a matrix of days for a month. Thus, a language that is used by non IT designers should have the following properties:

- Graphical representation: Boxes and arrows provide a better overview of control flow than plain programming syntax, but at the same time can be misleading and ambiguous.
- Programming constructs in the language should reflect real world objects

Support of Workflow Patterns: A systematic review of workflow patterns has been introduced in [166]. They are the means to show the expressiveness of a workflow language. As a simple example of a control flow pattern we would like to mention the sequence workflow pattern shown in Figure 6.18. This workflow pattern describes the ability of modeling two or more activities so that they are executed in a sequence.

In Appendix A the reader can find more details about the workflow patterns used to evaluate these languages, along with the different scores each language received. In this list the chosen workflow languages are evaluated by their support of the workflow patterns. To compare the support of workflow patterns, we introduced a point-based system. If a workflow language supports a workflow pattern out of the box it gets two points for that workflow pattern. If the workflow pattern can be realized using combinations of other workflow patterns, then the workflow language gets one points. If the workflow pattern cannot be realized with a workflow language it get zero points. Certain workflow patterns that are more important can get a weight. This weight is shown in a separate column. In our examination all workflow patterns receive exactly the same weight. More specifically, we compare these languages based on 43 control-flow patterns and on 40 data based routing patterns. A detailed description of each pattern can be found in [141] and in [140] respectively. The following list provides a brief description of only a couple of those patterns since their functionality is particularly noteworthy for modeling processes in telecommunication domains:

- **Milestone**: The milestone workflow pattern defines that a task is enabled if the workflow is in a specific state.
- **Transient Trigger**: A workflow language implementing transient trigger behavior is able to handle a trigger event directly when it occurs. If the event cannot be immediately handled, it is deleted. Transient triggers are useful in telecommunication settings where certain events may occur that must be handled immediately. An example is the call-waiting event that may occur during a call.
- **Generalized And-Join**: The generalized and-join is a construct of a workflow language where each input trigger of a preceding process threat must be enabled to enable a subsequent execution threat. This is useful in a telecommunication setting when a telephone conference is established. The conference is opened when all partners have dialed in.
- **Thread Split**: A workflow language implementing the thread split workflow pattern is able to initiate a number of threads in the same process instance. This is useful when trying to define where a telephone conference is broadcasted to several third parties.
- **Task Data:** The task data workflow pattern defines that a set of data is only visible for one task. This may avoid dirty reads and writes on a specific set of data.

- **Task Post condition**: A task can only be completed if a post condition is true. This post condition depends on the data a task produces. This is useful in the finalizing phase of a phone call. The phone call has been finished correctly when a certain set of data has been produced by the services that enacted the call.

- **Dynamic Binding of Services**: The term dynamic binding of services describes a mechanism to bind services a workflow uses when the execution has reached the point where the service must be invoked. This mechanism uses constraints to select the best fitting service for the current situation. A workflow engine in a telecommunication setting must be able to react to changing situations in real time. Thus, a dynamic binding of services at runtime must be possible.

6.2.2 Comparison results

In this section we present the results of the comparison of the three languages. We examine the languages with regards to the aforementioned requirements. For all languages, the language itself and possible implications on the execution framework are taken into consideration as well.

SCaLE

Abstraction: SCaLE has been developed with focus on abstraction. The graphical language syntax supports the development of business process models by people not familiar with requirements for workflows within the telecommunications domain. The language constructs used in SCaLE can be used without the knowledge of the infrastructure implementing each action. Another aspect implemented in SCaLE is the abstraction of control flow. In SCaLE it is not necessary to define a certain control flow between the activities. Instead, data dependencies can be used to describe when an activity should start. Even though it is possible to create compositions in SCaLE that are not deadlock-free, it is possible to detect deadlocks and warn the designer when that happens. There are no shortcomings in SCaLE with regards to abstraction.

Flexibility: SCaLE implements dynamic service selection, late binding of services mechanisms and complex message exchanges. The shortcomings of SCaLE in terms of flexibility are the lack of an extension mechanism; SCaLE was intentionally designed to be able to acquire new functionality by means of accessing external services, in order to keep the language as simple as possible, unlike BPMN 2.0 that supports a very rich extension mechanism. Moreover SCaLE does not contain specific constructs for compensation; instead compensation can be implemented in SCaLE by means of event handling. Explicit correlation is also not available in SCaLE even though implicit correlation is possible by means of sessions.

Support for the integration of services with different interfaces using different communication protocols: With SCaLE's composition framework it is possible to integrate services using different protocols and interfaces using the adaptation layer of the execution agents. In order to integrate a new type of service with the composition framework a new execution agent has to be written that is responsible for transforming the message format of the new service to the internal message format of the composition framework. When the interface or communication protocol of an existing service is changed only the corresponding execution agent of the composition framework needs to be adapted to the change, leaving the rest of the components untouched.

Usability of the language for business process developers without an IT-background: SCaLE comes with a graphical language. The language uses boxes and arrows to represent activities and control flow. However, the chosen notation is quite abstract. Thus, it is not possible in the first place to see which box is the equivalent for which real world object.

Support of workflow patterns: With 83 points, SCaLE gets the same amount of points as BPMN 2.0.

BPEL: BPEL is a workflow language that has been designed to work in a Web service environment. BPEL's main focus is on the execution of long running business processes.

Abstraction: It is claimed that BPEL does not permit the creation of business processes that may lead to deadlocks. However, to the best of our knowledge, there is no formal proof that validates this claim [124]. Despite this fact, BPEL contains mechanisms to ensure that every process model terminates in the end. Shortcomings: The only technologies BPEL workflows can use to interact with services are Web service interfaces (WSDL/SOAP). Moreover, with BPEL it is possible to abstract from the control flow. But this needs strong knowledge of the language because rather complex constructs have to be designed to accomplish this goal. BPEL does not have different user modes to support the creation of simple business processes.

Flexibility: BPEL allows for flexible selection of services during the runtime of a business process. However, this feature is not a first class citizen of the language itself, it can be implemented implicitly by making query like requests (similar to the one's possible with SCaLE) to external repositories. Furthermore, has an extension mechanism, which makes it possible to introduce new kinds of activities in a BPEL process.

Support for the integration of services with different interfaces using different communication protocols: BPEL is built on top of the Web services layer. The use of a standardized interface description language, WSDL, is a limitation that makes it possible for people to integrate only with a very limited class of services. To overcome this limitation, wrappers are required around legacy systems in order to expose them through Web service interfaces.

Usability of the language for business process developers without an IT-background: BPEL has no defined graphical syntax. Thus in contrast to SCaLE it is harder to learn for business process developers with no IT-background.

Support of workflow patterns: With 87 points, BPEL the highest amount of points in this evaluation.

BPMN 2.0: BPMN formerly has been a graphical workflow language that lacked execution semantics. With version 2.0 BPMN acquired execution semantics and an XML basis.

Abstraction: The language constructs of BPMN 2.0 are not dependent on technologies used to execute a business process model written in BPMN 2.0. The shortcomings of BPMN are the absence of mechanisms to define cycle free and deadlock free business process models. Another shortcoming found in BPMN is that it can be used to model processes that cannot be executed by a computer, unlike

SCaLE and BPEL which both have a focus on processes that can be executed by a computer system.

Flexibility: BPMN 2.0 may be the most flexible workflow language in our examination. It is extensible, it can handle events, compensations, and it is equipped with a correlation mechanism. There are not shortcomings in BPMN 2.0 in terms of flexibility. However, the extensibility of BPMN 2.0 is limited to the attachment of attributes and elements to existing BPMN 2.0 elements.

Support for the integration of services with different interfaces using different communication protocols: Like BPEL, BPMN is limited to be executed in a Web service environment (see evaluation of BPEL).

Usability of the language for business process developers without an IT-background: BPMN comes with a graphical representation of the language. Boxes denote activities and arrows shown the control flow. Apart from that BPMN has several further graphical elements to influence the control flow of a workflow. However, boxes and arrows are rather abstract. It is not immediately possible to see which box is executed on which service.

Support of workflow patterns: With 83 points, BPMN 2.0 gets the same amount of points as SCaLE.

SCaLE is the strongest language in our comparison with regards to abstraction, since it was conceived with option in mind by design; it permits the designer to focus on what actions are needed in order to achieve a goal and how such actions communicate based on data dependencies. However, SCaLE is not as strong as BPMN 2.0 with regards to extensibility since it cannot be extended at the language level, instead SCaLE enjoys extensibility at the composition framework level that can always be extended very easily to support new technologies in order to communicate with external functionality. The weakest language from a flexibility perspective is BPEL since it does not allow for extensions to be made at the linguistic level, or from the perspective of its underlying execution environment, leaving the BPEL isolated in the Web service space. However, BPEL is the language to score the highest amount of points for supporting the largest number of workflow patterns. SCaLE on the other hand scored as high as BPMN 2.0.

6.3 Summary

This chapter provided an evaluation of our work first by evaluating the proposed composition framework and thereafter by evaluating SCaLE. By evaluating our composition framework our goal was to show that we have developed a framework, that provides close to soft-real time responsiveness and that can handle a heavy load of requests. This has been done by measuring the overhead (delay) introduced by our composition framework as opposed to a simpler environment were service invocation is implemented without the composition framework. Our results show that the overhead introduced by the composition framework is very small: 4 to 12 msec. In addition, we have done extensive testing, with regards to throughput in order to show that our fine-grained composition framework provides better throughput ranging from +3% to +49% for moderate load and no upper limit for higher load, as opposed to the throughput provided by other coarse-grained rivals. With regards to response time, the fine-grained approach has a range of -8% to -33% for HTTP as opposed to coarse-grained approach deployed within a container. Another interesting point to note here is the robustness of the proposed composition framework under heavy load. Our experiments have shown that the proposed framework is stable after running for a long period, having received a large amount of requests.

Regarding the comparison between the different configurations of fine-grained composition core, we were able to verify our expectations:

- Coarse-grained approaches queue up requests because of blocking during I/O operations (e.g., external service invocations, database queries for the purposes of service discovery). This may lead to resource starvation and strong interference between composite applications when too many of them are blocked.
- The proposed fine-grained composition core never blocks, thanks to the non-blocking invocation of external services. Particularly, when the core is configured to run in asynchronous mode and standalone deployment, it provides better throughput (requests per second) and lower end-to-end response time.

A secondary dimension evaluated as a consequence of the empirical tests is that of stability. More specifically the prototype version was able to sustain robustly without demonstrating memory leaks back-to-back tests with approximately 30000 requests per test session.

Our next evaluation targets a qualitative comparison of SCaLE with its two closest rivals, WS-BPEL and BPMN 2.0. The qualitative evaluation compares these languages in terms of abstraction, flexibility, support for the integration of services with different interfaces using different communication protocols, usability of the language for business process developers without an IT-background and finally support of workflow patterns.

Unlike BPMN and WS-BPEL, SCaLE is deprived of constructs for parallel execution of activities within a composition. Instead, all actions in SCaLE are executed concurrently by default. Optionally, if synchronous execution order needs to be imposed, that is also possible using special constructs. Moreover, it supports dynamic binding of services at run time (DSS). Compositions in SCaLE are comprised of atomic and composite actions. Atomic actions are responsible for executing one function and afterwards provide the result of the computation of that function. Composite actions are actions that can contain further composite or atomic actions. In SCaLE the execution order of actions is defined using data dependencies and events. Data dependencies connect data outputs and data inputs of actions. Hence, if an action provides data at the output channel the trigger is enabled and the target action of that trigger is invoked. Events can come from the underlying composition framework. Due to the fact that SCaLE has a graphical notation it is easier for designers with non-IT workflow background to learn and comprehend the language and build workflows.

Along side with SCaLE a composition framework is utilized to interpret a composition at runtime. The composition framework provides support for the integration of services using different protocols and interfaces using a set of execution agents. With this set of execution agents it is possible to integrate new services, thus extending the composition framework without touching its composition core. If the specification for the interface to an external service changes, one only needs to modify the corresponding execution agent.

Our qualitative evaluation shows that SCaLE is the strongest language in terms of abstraction, liberating the designer from the burden of identifying how different actions can be executed in parallel. The main drawback identified in the comparison is SCaLE's lack of extensibility at the language level; SCaLE can only be extended at the composition framework level through execution agents. Last but not least, with regards to workflow pattern support, SCaLE rates as high as BPMN 2.0.

Chapter 7

Conclusions

The paradigm of service composition aims at allowing a designer to develop new services by shifting her attention more towards what the expected service should do and less towards on how that is done. Ultimately, this shift should lead to lower implementation costs and higher reliability in service development. In this thesis, after providing a classification of the approaches that could be used for the process of service composition, we presented our approach towards service composition. We were able to separate the limitations/challenges found in the state of the art in two main levels, one that dealt with the underlying execution frameworks and the other that dealt with languages used to express service compositions. This separation allowed us to tackle the issues encountered in each level in isolation and propose solutions for them.

A major contribution of this thesis is SCaLE — a graphical language for heterogeneous service composition. Unlike BPMN 2.0 and WS-BPEL, two very prominent and widely used languages that are used to describe workflows and to some extend compositions, SCaLE is deprived of constructs for parallel execution of actions. Instead, all actions within SCaLE are executed concurrently by default. Moreover, it supports dynamic service selection of services at run time. Compositions in SCaLE are comprised of atomic and composite actions. Atomic actions are responsible for executing one function and afterwards provide the result of the computation of that function. Composite actions are actions that can contain nested composite or atomic actions. In SCaLE the execution flow of a service is defined using data dependencies and events. Data dependencies connect data outputs and data inputs of actions.

Events can originate from the underlying composition framework. To avoid possible race conditions, only copies of variables are being transferred between actions in order to avoid changing the original ones. The original value of a variable can only be changed by special variables known as effects. Due to the fact that SCaLE has a graphical notation it is easier for designers with non-IT workflow background to learn and comprehend the language and build compositions. In addition, SCaLE has formally specified execution semantics. Even though it is possible to implement deadlocking compositions with SCaLE, it is possible to detect them in advance due to the graph like nature of the language. Our qualitative evaluation has shown that the expressiveness of SCaLE with regards to workflow patterns, is tantamount to that of BPMN 2.0.

The second contribution of this thesis is an asynchronous, event-driven non-blocking composition framework. The composition framework possesses the following characteristics:

- integration of heterogeneous service technologies within a single composite application
- optional type system
- service interaction
- full control of execution flow through the use of composite application skeletons described using SCaLE

Within the premises of the composition framework, the problem of dealing with heterogeneous services is dealt by the convention of technologic specific execution agents, thereby permitting the core to be technology agnostic. Our empirical evaluations have shown that our composition framework is scalalable; limited by the amount of available memory and not by the amount of processing threads. Moreover we have found that the overhead introduced by the composition framework is very small: 4 to 12msec. The composition framework has been developed by utilizing a programming style known as CPS that allowed for implementing a fine-grained composition core, capable of splitting the process of interpreting a composite skeleton into phases. This transformation allowed for higher throughput ranging from +3% to +49% for moderate load as opposed to rival coarse-grained approaches.

These contributions were made concrete, by the design and implementation of the proposed composition framework and the proposed language for service composition. The composition technology described in this thesis has shown its potential in real-life use cases [145, 78] and is already available in commercial deployments. Moreover, in [169, 170] we illustrate two use cases, one in the cross Telco — Web 2.0 and one in the area of machine-to-machine communication that have been implemented using the proposed technology. Last but not least, in [120] we show that extensible workflow languages such as BPMN 2.0 can be augmented to support some of the constructs available in SCaLE such as Dynamic Service Selection (action service template), start conditions for actions, event filter tasks and tasks for communicating with execution agents, thereby showing that our contributions are generic.

Chapter 8

Future Work

In our view, service composition is an interesting research area as such, since it revisits the concept of writing and implementing software by abstracting away from the machinery responsible for executing a particular piece of software. Instead, service composition focuses more closely on what the designer has in mind for the particular service and not on how that is done. This shift in focus becomes evident when one observes the set of constructs used in pure languages for service composition such as SCaLE as opposed to workflow languages that can be used for expressing service compositions. It is evident from the small amount of constructs that are available to the designer that she needs to firmly embrace a discipline of re-using pre-existing functionality instead of re-inventing or implementing constructs from scratch. Therefore, the key characteristic that has marked the success or failure of any given programming language in history, that of the amount of supporting libraries with pre-existing functionality, becomes quintessential for the success of service composition languages, along with an increase in the set of external technologies that can be accessed by underlying composition frameworks. In addition, to these remarks, this section iterates over a brief set of new features we see of interest. The proposed set means in no way to limit the possibilities for further work in this field, but rather to provide a subtle push towards some interesting directions, by taking into consideration current trends.

8.1 Multi tenancy support in a cloud deployment

Multi-tenancy support in a cloud deployment has two facets — one with regards to the underlying composition framework and the other with regards to the designer's implementing the compositions. From the underlying composition framework's point of view, it is interesting to examine the means and mechanism for democratizing the utilization per tenant. Figuring out techniques that allow for measuring the amount of resources that are occupied per composition and making sure that a tenant is not hogging all resources for themselves. From the point of view of the designer's, assuming a cloud deployment one design and implement interesting recommendation engines that can provide predictions on what the next element in the process of designing a service composition can be, simple by iterating over the collected data that our store in a repository in the cloud of previously created service compositions. Such a tool could be very useful for new users of service composition and could possible cut the learning curve of understanding service composition languages.

8.2 Domain-specific language for Big data analytics

The process of service composition appears to be applicable in the area of big data analytics where languages such as R [161], Mathematica [176] and Matlab [72] are being used to construct very specific patterns for analyzing big data. These patterns are rather rigid, since in such languages one needs to explicitly define the flow of the analysis and the corresponding format of data exchanges and the association between data and each external library that does the analytical computation (such as classification, cluster analysis, ensemble learning, neural networks, pattern recognition, anomaly detection, predictive modeling, regressing, sentiment analysis and others). It would be interesting to apply languages such as SCaLE in the process of data analytics in data intensive scenarios, along with mechanisms for Complex Event Processing (CEP) [21] in order to leverage both, the potential of multi-core hardware, but also to make it easy for a designer to construct analytic patterns.

Bibliography

[1] Active Endpoints. Activebpel engine, 2008.

[2] A. J. Albrect, J. Gaffney, and E. John. Software function, source lines of code, and development effort prediction: a software science validation. *Software Engineering, IEEE Transactions on*, pages 639–648, 1982.

[3] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 1015–1022. ACM, 2009.

[4] A. Alexandrescu. *The D programming language*. Addison-Wesley Professional, 2010.

[5] A. Almossawi. How maintainable is the firefox codebase?, 2013. URL `http://almossawi.com/firefox/prose/`.

[6] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Computing Surveys (CSUR)*, 40, 2008.

[7] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, and T. Payne. Daml-s: Web service description for the semantic web. *The Semantic Web—ISWC 2002*, pages 348–363, 2002.

[8] Anónimo, TPCI. Tiobe programming community index, 2006.

[9] C. Arad, J. Dowling, and S. Haridi. Developing, simulating, and deploying peer-to-peer systems using the kompics component model. In *Proceedings of the Fourth International ICST Conference on Communication system software and middleware*, page 16. ACM, 2009.

[10] J. Armstrong, R. Virding, C. Wikstrom, and M. Williams. *Concurrent programming in ERLANG*. Prentice Hall, 1996.

[11] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787–2805, 2010.

[12] C. Y. Baldwin, A. D. MacCormack, and J. Rusnak. Hidden structure: Using network methods to map product architecture. *Harvard Business School Working Paper, No. 13–093*, 2013. URL `http://nrs.harvard.edu/urn-3:HUL.InstRepos:10646422`.

[13] M. Bali. *Drools JBoss Rules 5.0 Developer's Guide*. Packt Publishing Ltd, 2009.

[14] M. Ben-Ari. *Principles of concurrent and distributed programming*. Pearson Education, 2006.

[15] D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic service composition based on behavioral descriptions. *International Journal of Cooperative Information Systems*, 14(04):333–376, 2005.

[16] G. Bertrand. The ip multimedia subsystem in next generation networks. *Network, Multimedia and Security department (RSM)-GET/ENST Bretagne*, 2007.

[17] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in cω. In *ECOOP 2005-Object-Oriented Programming*, pages 287–311. Springer, 2005.

[18] G. Bond, E. Cheung, I. Fikouras, and R. Levenshteyn. Unified telecom and web services composition: problem definition and future directions. In *Proceedings of the 3rd International Conference on Principles, Systems and Applications of IP Telecommunications*, page 13. ACM, 2009.

[19] J. Bredereke. Maintaining telephone switching software requirements. *Communications Magazine, IEEE*, 40:104–109, 2002.

[20] K. Brown. *Enterprise Java Programming with IBM WebSphere, disk 1*. Addison-Wesley Professional, 2003.

[21] A. Buchmann and B. Koldehofe. Complex event processing. *it-Information Technology*, 51(5):241–242, 2009.

[22] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the 12th international conference on World Wide Web*, pages 403–410. ACM, 2003.

[23] M. Cadoli and M. Lenzerini. The complexity of propositional closed world reasoning and circumscription. *Journal of Computer and System Sciences*, 48: 255–310, 1994.

[24] P. R. Calhoun, G. Zorn, P. Pan, and H. Akhtar. Diameter framework document. *Work in Progress*, 2001.

[25] G. Camarillo and M.-A. Garcia-Matrin. *The 3G IP multimedia subsystem (IMS): merging the Internet and the cellular worlds.* Wiley, 2007.

[26] L. Cardelli. Type systems. *ACM Computing Surveys*, 28:263–264, 1996.

[27] J. Cardoso and A. Sheth. Semantic e-workflow composition. *Journal of Intelligent Information Systems*, 21:191–225, 2003.

[28] F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M.-C. Shan. Adaptive and dynamic service composition in eflow. In *Advanced Information Systems Engineering*, pages 13–31. Springer, 2000.

[29] D. A. Chappell. *Enterprise service bus.* O'Reilly media, 2009.

[30] A. Chen. *Context-aware collaborative filtering system: Predicting the user's preference in the ubiquitous computing environment*, pages 244–253. Springer, 2005.

[31] L. Chen, N. R. Shadbolt, C. Goble, F. Tao, S. J. Cox, C. Puleston, and P. R. Smart. Towards a knowledge-based approach to semantic service composition. In *The Semantic Web-ISWC 2003*, pages 319–334. Springer, 2003.

[32] Z. Cheng, M. P. Singh, and M. A. Vouk. Composition constraints for semantic web services. In *WWW2002 Workshop on Real World RDF and Semantic Web Applications*, 2002.

[33] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. In *ECOOP 2002—Object-Oriented Programming*, pages 231–255. Springer, 2006.

[34] C. Chrighton, D. Long, and D. Page. Jain slee vs sip servlet which is the best choice for an ims application server? In *Telecommunication Networks and Applications Conference, 2007. ATNAC 2007. Australasian*, pages 448–453. IEEE, 2007.

[35] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, et al. Web services description language (wsdl) 1.1, 2001.

[36] B. A. Christudas. *Service Oriented Java Business Integration: Enterprise Service Bus Integration Tions for Java Developers.* Packt Pub Limited, 2008.

[37] S. A. Chun, V. Atluri, and N. Adm. Using semantics for policy-based web service composition. *Distributed and Parallel Databases*, 18:37–64, 2005.

[38] B. Collins-Sussman. The subversion project: buiding a better cvs. *Linux Journal*, 2002(94):3, 2002.

[39] K. Cooper and L. Torczon. *Engineering a compiler.* Morgan Kaufmann, 2011.

[40] M. Cumberlidge. *Business Process Management with JBoss JBPM: A Practical Guide for Business Analysts; Develop Business Process Models for Implementation in a Business Process Management Sytem.* Packt Publishing, 2007.

[41] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *Internet Computing, IEEE*, 6:86–93, 2002.

[42] F. Curbera, F. Leymann, T. Storey, D. Ferguson, and S. Weerawarana. *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more.* Prentice Hall PTR Englewood Cliffs, 2005.

[43] E. Dahlman, S. Parkvall, J. Skold, and P. Beming. *3G evolution: HSPA and LTE for mobile broadband.* Access Online via Elsevier, 2010.

[44] T. Dinsing, G. A. Eriksson, I. Fikouras, K. Gronowski, R. Levenshteyn, P. Pettersson, and P. Wiss. Service composition in ims using java ee sip servlet containers. *Ericsson Review*, 3:92–96, 2007.

[45] L. Dryburgh and J. Hewett. *Signalling System No. 7 (SS7/C7): Protocol, Architecture, and Services.* Cisco press, 2005.

[46] D. F. D'souza and A. C. Wills. *Objects, components, and frameworks with UML: the catalysis approach*, volume 1. Addison-Wesley Reading, 1998.

[47] S. Dustdar and W. Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1:1–30, 2005.

[48] E. ECMAScript, E. C. M. Association, et al. Ecmascript language specification, 2011.

[49] A. Egloff. Ajax push (aka comet) with java™ business integration (jbi). In *Presentation at the JavaOne 2007 Conference, San Francisco, CA*, volume 5, 2007.

[50] W. Emmerich, B. Butchart, L. Chen, B. Wassermann, and S. L. Price. Grid service orchestration using the business process execution language (bpel). *Journal of Grid Computing*, 3:283–304, 2005.

[51] Ericsson LM. More than 50 billion connected devices, February 2011.

[52] T. Erl. *Service-oriented Architecture: Concepts, Technology, and Design*. Pearson Education India, 2006.

[53] V. Ermolayev and N. Keberle. Towards a framework for agent-enabled semantic web service composition. *International Journal of Web Services Research (IJWSR)*, 1:68–87, 2004.

[54] V. Ferraro-Esparza, M. Gudmandsen, and K. Olsson. Ericsson telecom server platform 4. *Ericsson Review*, 3:104–113, 2002.

[55] D. Ferry. Jain slee (jslee) 1.1 specification, final release, 2008.

[56] K. Figl, J. Mendling, M. Strembeck, and J. Reckner. *On the cognitive effectiveness of routing symbols in process modeling languages*, pages 230–241. Springer, 2010.

[57] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5: 345, 1962.

[58] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19:17–37, 1982.

[59] M. Fowler, R. Parsons, and J. MacKenzie. Pojo: An acronym for: Plain old java object, 2000, 2009.

[60] K.-S. Fu and B. K. Bhargava. Tree systems for syntactic pattern recognition. *Computers, IEEE Transactions on*, 100:1087–1099, 1973.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Abstraction and reuse of object-oriented design.* Springer, 2001.

[62] J. J. Garrett et al. Ajax: A new approach to web applications, 2005.

[63] R. Gayraud and O. Jacques. Sipp. *SIPp Tool (http://sipp. sourceforge. net)*, 2004.

[64] J. C. Giarratano and G. Riley. *Expert systems.* PWS Publishing Co., 1998.

[65] G. K. Gill and C. F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *Software Engineering, IEEE Transactions on*, 17: 1284–1288, 1991.

[66] M. Girdley, S. L. Emerson, and R. Woollen. *J2EE Applications and BEA WebLogic Servers.* Prentice Hall PTR, 2001.

[67] G. Gomez and R. Sanchez. End-to-end quality of service over cellular networks. *England, John Wiley and Sons*, 2005.

[68] A. Gouya, N. Crespi, and E. Bertin. Scim (service capability interaction manager) implementation issues in ims service architecture. In *Communications, 2006. ICC'06. IEEE International Conference on*, volume 4, pages 1748–1753. IEEE, 2006.

[69] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete mathematics: a foundation for computer science*, volume 2. Addison-Wesley Reading, 1989.

[70] R. Gronmo, D. Skogan, I. Solheim, and J. Oldevik. Model-driven web services development. In *e-Technology, e-Commerce and e-Service, 2004. EEE'04. 2004 IEEE International Conference on*, pages 42–45. IEEE, 2004.

[71] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, A. Karmarkar, and Y. Lafon. Soap version 1.2. *W3C recommendation*, 24, 2003.

[72] M. U. Guide. The mathworks. *Inc., Natick, MA*, 5, 1998.

[73] T. Gunarathne, D. Premalal, T. Wijethilake, I. Kumara, and A. Kumar. Bpel-mora: lightweight embeddable extensible bpel engine. In *Emerging Web Services Technology*, pages 3–20. Springer, 2007.

[74] H. Haas and A. Brown. Web services glossary. *W3C Working Group Note (11 February 2004)*, 2004.

[75] M. J. Hadley. Web application description language (wadl), 2006.

[76] E. H. Halili. *Apache JMeter: A Practical Beginner's Guide to Automated Testing and performance measurement for your websites.* Packt Publishing Ltd, 2008.

[77] R. Hamadi and B. Benatallah. A petri net-based model for web service composition. In *Proceedings of the 14th Australasian database conference-Volume 17*, pages 191–200. Australian Computer Society, Inc., 2003.

[78] J. Hammil. Ericsson and vennetics successfully integrate j-box with the ericsson composition engine, 2009.

[79] M. Handley and V. Jacobson. Session description protocol. *Request for Comments*, 2327, 1998.

[80] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java message service. *Sun Microsystems Inc., Santa Clara, CA*, 2002.

[81] R. Hazan and P. Andruszkiewicz. Home pages identification and information extraction in researcher profiling. In *Intelligent Tools for Building a Scientific Information Platform*, pages 41–51. Springer, 2013.

[82] M. Heffernan and K. Wilken. Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling*, 8:427–451, 2005.

[83] I. Hickson. The websocket protocol. *draft-ietf-hybi-thewebsocketprotocol-00 (work in progress)*, 2010.

[84] D. Hornford. Definition of soa. *The Open Group*, 2006.

[85] I. Horrocks. Daml+oil: A description logic for the semantic web. *IEEE Data Engineering Bulletin*, 25:4–9, 2002.

[86] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *Software Engineering, IEEE Transactions on*, 24:831–847, 1998.

[87] E. Jendrock. *The Java EE 5 Tutorial: For Sun Java System Application Server Platform Edition 9*. Addison-Wesley Professional, 2006.

[88] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. *Functional programming languages and computer architecture*, pages 190–203, 1985.

[89] M. B. Juric. *Ws-Bpel 2.0 for Soa Composite Applications with Oracle Soa Suite 11G*. Packt Publishing, 2010.

[90] D. Kitchin, A. Quark, W. Cook, and J. Misra. The orc programming language. In *Formal Techniques for Distributed Systems*, pages 1–25. Springer, 2009.

[91] D. Kohlert and A. Gupta. The java api for xml-based web services (jax-ws) 2.1, 2007.

[92] D. Kovacs. Bnf definition of pddl 3.1. *Unpublished manuscript from the IPC-2011 website*, 2011.

[93] J. R. Koza, I. Bennett, H. Forrest, and O. Stiffelman. *Genetic programming as a Darwinian invention machine*. Springer, 1999.

[94] A. Kristensen et al. Jsr-116: Sip servlet api. *Java Community Process*, 2003.

[95] A. Lakshman and P. Malik. The apache cassandra project, 2011.

[96] I. Lee and R. K. Iyer. Faults, symptoms, and software fault tolerance in the tandem guardian90 operating system. In *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pages 20–29. IEEE, 1993.

[97] M. Lekavỳ and P. Návrat. Expressivity of strips-like and htn-like planning. In *Agent and Multi-Agent Systems: Technologies and Applications*, pages 121–130. Springer, 2007.

[98] H. J. Levesque, R. Reiter, Y. Lesperance, F. Lin, and R. B. Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31:59–83, 1997.

[99] S. Liang, P. Fodor, H. Wan, and M. Kifer. Openrulebench: an analysis of the performance of rule engines. In *Proceedings of the 18th international conference on World wide web*, pages 601–610. ACM, 2009.

[100] T. Lindholm and F. Yellin. *Java virtual machine specification.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[101] R. Love. *Linux kernel development.* Pearson Education, 2010.

[102] A. MacCormack. The architecture of complex systems: Do core-periphery structures dominate? In *Academy of Management Proceedings*, volume 2010, pages 1–6. Academy of Management, 2010.

[103] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52:1015–1030, 2006.

[104] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, and T. Payne. Owl-s: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.

[105] P. McCarthty and D. Crane. *Comet and Reverse Ajax: The Next-Generation Ajax 2.0.* Apress, 2008.

[106] J. McCarthy and P. Hayes. *Some philosophical problems from the standpoint of artificial intelligence.* Stanford University, 1968.

[107] D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. Pddl-the planning domain definition language, 1998.

[108] D. V. McDermott. Estimated-regression planning for interactions with web services. In *AIPS*, volume 2, pages 204–211, 2002.

[109] D. L. McGuinness and F. Van Harmelen. Owl web ontology language overview. *W3C recommendation*, 10:2004–03, 2004.

[110] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. *KR*, 2:482–493, 2002.

[111] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal—The International Journal on Very Large Data Bases*, 12:333–351, 2003.

[112] L. Mei, W. K. Chan, and T. Tse. An adaptive service selection approach to service composition. In *Web Services, 2008. ICWS'08. IEEE International Conference on*, pages 70–77. IEEE, 2008.

[113] S. Meloan. The java hotspot performance engine: An in-depth look. *Sun Microsystems, Jun*, 1999.

[114] R. Merkel. Managing your money with gnucash. *Linux Journal*, 2001(84es):8, 2001.

[115] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[116] MySQL AB. Mysql, 2001.

[117] S. Narayanan and S. A. McIlraith. Simulation, verification and automated composition of web services. In *Proceedings of the 11th international conference on World Wide Web*, pages 77–88. ACM, 2002.

[118] Neo4J Developers. Neo4j. *Graph NoSQL Database [online]*, 2012.

[119] J. Niemoller, E. Freiter, K. Vandikas, R. Quinet, R. Levenshteyn, and I. Fikouras. Multi-technology service composition for the telecommunication domain-concepts and experiences. In *Next Generation Mobile Applications, Services and Technologies (NGMAST), 2010 Fourth International Conference on*, pages 34–41. IEEE, 2010.

[120] J. Niemoller, K. Vandikas, R. Levenshteyn, D. Schleicher, and F. Leymann. Towards a service composition language for heterogeneous service environments. In *Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference on*, pages 121–126. IEEE, 2011.

[121] M. Odersky. The scala language specification, version 2.8. *EPFL Lausanne, Switzerland*, 2009.

[122] Oracle. Oracle bpel process manager, 2009.

[123] OrientDB Developers. Orientdb. *Hybrid Document-Store and Graph NoSQL Database [online]*, 2012.

[124] C. Ouyang, E. Verbeek, W. M. van der Aalst, S. Breutel, M. Dumas, H. Ter, and H. Arthur. Formal semantics and analysis of control flow in ws-bpel. *Science of Computer Programming*, 67:162–198, 2007.

[125] M. Pantazoglou, A. Tsalgatidou, and G. Athanasopoulos. Discovering web services and jxta peer-to-peer services in a unified manner. In *Service-Oriented Computing–ICSOC 2006*, pages 104–115. Springer, 2006.

[126] C. Papadimitriou and M. Sideri. On the floyd–warshall algorithm for logic programs. *The journal of logic programming*, 41:129–137, 1999.

[127] C. H. Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[128] M. P. Papazoglou. Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pages 3–12. IEEE, 2003.

[129] R. E. Park. Software size measurement: A framework for counting source statements. Technical report, DTIC Document, 1992.

[130] A. A. Patil, S. A. Oundhakar, A. P. Sheth, and K. Verma. Meteor-s web service annotation framework. In *Proceedings of the 13th international conference on World Wide Web*, pages 553–562. ACM, 2004.

[131] M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI*, pages 1252–1259, 2005.

[132] G. D. Plotkin. A structural approach to operational semantics, 1981.

[133] J. Rao and X. Su. *A survey of automated web service composition methods*, pages 43–54. Springer, 2005.

[134] C. Reis, A. Barth, and C. Pizano. Browser security: lessons from google chrome. *Queue*, 7(5):3, 2009.

[135] D. Riehle and T. Gross. *Role model based framework design and integration*, volume 33, pages 117–133. ACM, 1998.

[136] J. Robinson, I. Wakeman, and T. Owen. Scooby: middleware for service composition in pervasive computing. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 161–166. ACM, 2004.

[137] M. Rodrigez. Mysql vs. neo4j on a large-scale graph traversal, 2010.

[138] M. A. Rodriguez et al. Gremlin, 2011.

[139] A. L. Rubinger and B. Burke. *Enterprise JavaBeans 3.1*. O'Reilly Media, 2010.

[140] N. Russell, A. H. Ter Hofstede, D. Edmond, and W. M. van der Aalst. Workflow data patterns: Identification, representation and tool support. In *Conceptual Modeling–ER 2005*, pages 353–368. Springer, 2005.

[141] N. Russell, A. H. Ter Hofstede, and N. Mulyar. Workflow controlflow patterns: A revised view, 2006.

[142] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*, volume 74. Prentice hall Englewood Cliffs, 1995.

[143] T. Russell. *Session Initiation Protocol (SIP): controlling convergent networks*. McGraw-Hill Osborne Media, 2008.

[144] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and symbolic computation*, 6:289–360, 1993.

[145] S. Samms. Intel solution brief: Integrating existing applications in the evolving telecommunications network, 2009.

[146] P. Sarang, K. Gabhart, D. Young, A. Tost, T. McAllister, R. Adatia, K. Shah, and J. Griffin. *Professional EJB*. Wrox Press Ltd., 2001.

[147] scitools.com. Understand, 2013.

[148] K. Scribner. *Microsoft Windows workflow foundation step by step*. Microsoft Press, 2009.

[149] H. Shutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30:202–210, 2005.

[150] SIP SailFin. Application server, 2007.

[151] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *Web services: modeling, architecture and infrastructure workshop in ICEIS*, volume 2003. Citeseer, 2003.

[152] E. Sirin, B. Parsia, D. Wu, J. Handler, and D. Nau. Htn planning for web service composition using shop2. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1:377–396, 2004.

[153] P. G. Smith. Accelerated product development: techniques and traps. *The PDMA Handbook of New Product Development, KB Kahn, Editor. Wiley & Sons*, pages 173–187, 2004.

[154] B. Srivastava and J. Koehler. Web service composition-current solutions and open problems. In *ICAPS 2003 workshop on Planning for Web Services*, volume 35, pages 28–35, 2003.

[155] T. Steinmetz. Ein event-modell für ws-bpel 2.0 und dessen realisierung in apache ode, 2008.

[156] D. J. Sturtevant. *System design and the cost of architectural complexity.* PhD thesis, Massachusetts Institute of Technology, 2013.

[157] G. Sussman, H. Abelson, and J. Sussman. Structure and interpretation of computer programs. *The Massachusetts Institute of Technology*, 10, 1985.

[158] A. S. Tanenbaum. *Modern operating systems*, volume 2. prentice Hall Englewood Cliffs, 1992.

[159] A. S. Tanenbaum and A. S. Woodhull. *Operating systems: design and implementation*, volume 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

[160] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1:146–160, 1972.

[161] R. C. Team et al. R: A language and environment for statistical computing. *Vienna, Austria: R Foundation for Statistical Computing*, pages 1–1731, 2008.

[162] C. Tselios, H. Perkuhn, K. Vandikas, and M. Kampann. Targeted mobile advertisement in the ip multimedia subsystem. *Online Multimedia Advertising: Techniques and Technologies*, page 279, 2011.

[163] M. T. Tut and D. Edmond. *The use of patterns in service composition*, pages 28–40. Springer, 2002.

[164] A. Van Der, M. Wil, H. Ter, and H. Arthur. Yawl: yet another workflow language. *Information systems*, 30:245–275, 2005.

[165] W. M. Van Der Aalst, H. Ter, H. Arthur, and M. Weske. *Business process management: A survey*, pages 1–12. Springer, 2003.

[166] W. M. van Der Aalst, A. H. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[167] W. M. Van der Aalst, M. Weske, and D. Grünbauer. Case handling: a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2): 129–162, 2005.

[168] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT, 2003.

[169] K. Vandikas, E. Freiter, R. Levenshteyn, R. Quinet, J. Niemoller, and I. Fikouras. Blending the telecommunication domain with web 2.0 services. In *Intelligence in Next Generation Networks (ICIN), 2010 14th International Conference on*, pages 1–6. IEEE, 2010.

[170] K. Vandikas, N. C. Liebau, M. Dohring, L. Mokrushin, and I. Fikouras. M2m service enablement for the enterprise. In *Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference on*, pages 169–174. IEEE, 2011.

[171] T. Velte, A. Velte, and R. Elsenpeter. *Cloud computing, a practical approach.* McGraw-Hill Inc., 2009.

[172] B.-F. Wang and G.-H. Chen. Constant time algorithms for the transitive closure and some related graph problems on processor arrays with reconfigurable bus systems. *Parallel and Distributed Systems, IEEE Transactions on*, 1:500–507, 1990.

[173] H. Wen. Abiword: Open source's answer to microsoft word. *Linux Dev Center, downloaded from http://www. linuxdevcenter. com/lpt/a/1636*, pages 1–3, 2002.

[174] S. A. White. Introduction to bpmn. *iBM Cooperation*, pages 2008–029, 2004.

[175] J. Wilkiewicz and M. Kulkarni. Jsr 289: Sip servlet specification v1. 1.

[176] S. Wolfram. *The MATHEMATICA® Book, Version 4*. Cambridge university press, 1999.

[177] M. Zim. Bringing soa value patterns to life. *Oracle Fusion Middleware*, 2006. URL `http://docs.oracle.com/cd/E23943_01/index.htm`.

[178] C. Zirpnins, W. Lamersdorf, and T. Baier. *Flexible coordination of service interaction patterns*, pages 49–56. ACM, 2004.

List of Figures

List of Tables

Glossary

3GPP Third Generation Partnership Project. 12, 14, 24, 25, 64

AI Artifical Intelligence. 30, 46, 47

AJAX Asynchronous JavaScript and XML. 49

AOP Aspect Oriented Programming. 166

AS Application Server. 15, 16

ASDL Agent Service Description Language. 35

BCSM Basic Call State Machine. 21

BPMN 2.0 Business Process Model and Notation 2.0. 27, 28, 45, 205–207

CAMEL Customized Applications for Mobile Networks Enhanced Logic. 49

CAP CAMEL Application Part. 56, 63

CAPv3 Camel Application Part Version 3. 22

CCN Charging Control Node. 22, 23

CDR Call Detail Records. 19, 22

CEP Complex Event Processing. 210

CPS Continuation Passing Style. 106, 153, 165, 169, 206

CPU Central Processing Unit. 42, 174, 175, 179, 190, 192

CSCF Call Session Control Function. 15, 127

IMS IP Multimedia Subsystem. 2, 3, 12, 14, 15, 18–20, 25, 26, 38, 49, 53, 63, 92, 94, 95, 97, 117, 129, 190

IN Intelligent Network. 15, 20, 21, 49, 98

IN EA IN Execution Agent. 63

INAP Intelligent Network Application Protocol. 56, 63

IP Internet Protocol. 12, 14, 15, 19, 56, 126

ISC IP Multimedia Service Control. 25, 63, 127, 129

JBI Java Business Integration. 58, 59, 63, 104

JBI EA JBI Execution Agent. 165

JSON JavaScript Object Notation. 113

JVM Java Virtual Machine. 104, 106, 175

LDAP Lightweight Directory Access Protocol. 25, 121, 123

LDIF LDAP Data Interchange Format. 121

MRFP Media Resource Function Processor. 135

MSC Mobile Switching Center. 18, 22

MSRP Message Session Relay Protocol. 135, 139

OMG Object Management Group. 28

P2P Peer-to-Peer. 135

PCRF Policy charging and rules functions. 15, 24, 36

PDDL Planning Domain Definition Language. 32

REST Representational state transfer. 2, 59, 104

S-CSCF Serving Call Session Control Function. 15, 19, 20, 126–130, 190

SASN Service Aware Support Node. 24

SCaLE Service Composition LanguagE. 5, 7, 8, 49, 51, 52, 65, 81, 86–88, 91, 97, 98, 101, 171, 193, 198–201, 203–207, 209, 210

SCE Service Creation Environment. 22

SCF Service Control Function. 21

SCIM Service Capability Interaction Manager. 15, 25, 26, 44, 64

SCP Service Control Point. 21, 22

SDF Service Data Function. 22

SDP Service Data Point. 22, 23, 120

SGSN Serving GPRS Support Node. 22

SIP Session Initiation Protocol. 4, 12, 14–20, 25, 26, 38–41, 56, 98, 102, 103, 105, 115, 117, 121–135, 139–148, 150, 157, 158, 164–166, 168, 169, 173, 189, 190, 192

SIP EA SIP Execution Agent. 63, 126

SMS Short Message Service. 22, 23

SOA Service Oriented Architecture. 1, 11, 20, 25, 50, 56

SOAP Simple Object Access Protocol. 1, 2, 10, 11, 26, 32, 65, 78, 88, 89, 104, 145, 169, 199

SRF Specialized Resource Function. 22

SS7 Service Control Signalling System Version 7. 22

SSF Service Switching Function. 21

SSP Service Switching Point. 21, 22

UDDI Universal Description, Discovery and Integration. 28, 29

UML Unified Modeling Language. 28, 29, 44, 45, 148

URI Uniform Resource Identifier. 20, 121, 130

USSD Unstructured Supplementary Service Data. 22, 23

VoIP Voice Over IP. 120, 121, 123, 124, 127, 128, 130, 135, 139

W3C World Wide Web Consortium. 10

WADL Web Application Description Language. 59, 146, 147

WS-BPEL Web Services Business Process Execution Language. 26, 27, 29, 45, 205

WSDL Web Services Definition Language. 4, 10, 11, 26, 29, 32, 35, 46, 49, 59, 120, 146, 147, 150, 199, 200

XML Extensible Markup Language. 11, 29, 88, 113, 200

YAWL Yet Another Workflow Language. 45

Appendix A

This appendix details the linguistic comparison we have made between BPEL, BPMN 2.0 and SCaLE, in terms of workflow pattern support. The comparison is split in two parts, each one represented by a table. Table 8.1 compares these languages in terms of supporting control flow patterns, while Table 8.2 focuses on data pattern support. By taking the sum of the individual points in each column from both tables, we arrive to the following results: BPEL: 87, BPMN 2.0: 83, SCaLE: 83

Control flow patterns	**BPEL**	**BPMN 2.0**	**SCaLE**
Sequence	2	2	2
Parallel Split	2	2	2
Synchronization	2	2	2
Exclusive Choice	2	2	2
Simple Merge	2	2	2
Multi-Choice	2	2	2
Structured Synchronizing Merge	2	2	2
Multi-Merge	2	2	2
Structured Discriminator	2	2	2
Arbitrary Cycles	0	0	2
Implicit Termination	2	2	2
Multiple Instances without Synchronization	2	2	2
Multiple Instances with a Priori Design-Time Knowledge	0	0	0
Multiple Instances with a Priori Run-Time Knowledge	0	0	0
Multiple Instances without a Priori Run-Time Knowledge	0	0	2
Deferred Choice	2	2	2
Interleaved Parallel Routing	1	1	1
Milestone	0	0	2
Cancel Activity	2	2	0
Cancel Case	2	2	0
Structured Loop	2	2	2
Recursion	0	0	2
Transient Trigger	0	0	2
Persistent Trigger	2	2	0
Cancel Region	1	1	0
Cancel Multiple Instance Activity	0	0	0
Complete Multiple Instance Activity	0	0	0
Blocking Discriminator	0	0	0
Cancelling Discriminator	2	2	0
Structured Partial Join	0	0	0
Blocking Partial Join	0	0	0
Cancelling Partial Join	0	0	0
Generalised AND-Join	2	2	2
Static Partial Join for Multiple Instances	0	0	0
Cancelling Partial Join for Multiple Instances	0	0	0
Dynamic Partial Join for Multiple Instances	0	0	0
Local Synchronizing Merge	2	2	2
General Synchronizing Merge	2	2	0
Critical Section	2	0	0
Interleaved Routing	2	1	0
Thread Merge	0	0	0
Thread Split	1	1	2
Explicit Termination	2	2	0

Table 8.1: Control flow evaluation

Data patterns	**BPEL**	**BPMN 2.0**	**SCaLE**
Task Data	2	2	2
Block Data	0	0	1
Scope Data	2	0	0
Multiple Instance Data	0	0	2
Case Data	2	2	0
Folder Data	0	0	0
Workflow Data	0	0	0
Environment Data	2	2	2
Task to Task	2	2	2
Block Task to SubWorkflow Decomposition	0	0	0
SubWorkflow Decomposition to Block Task	0	0	0
To Multiple Instance Task	0	0	0
From Multiple Instance Task	0	0	0
Case to Case	1	0	0
Task to Environment - Push-Oriented	2	2	2
Environment to Task - Pull-Oriented	2	2	2
Environment to Task - Push-Oriented	2	2	2
Task to Environment - Pull-Oriented	2	2	2
Case to Environment - Push-Oriented	0	0	0
Environment to Case - Pull-Oriented	0	0	0
Environment to Case - Push-Oriented	0	0	0
Case to Environment - Pull-Oriented	0	0	0
Workflow to Environment - Push-Oriented	0	0	0
Environment to Workflow - Pull-Oriented	0	0	0
Environment to Workflow - Push-Oriented	0	0	0
Workflow to Environment - Pull-Oriented	0	0	0
Data Transfer by Value - Incoming	2	2	2
Data Transfer by Value - Outgoing	2	2	2
Data Transfer - Copy In/Copy Out	1	1	1
Data Transfer by Reference - Unlocked	2	2	2
Data Transfer by Reference - With Lock	1	2	0
Data Transformation - Input	2	2	2
Data Transformation - Output	2	2	2
Task Precondition - Data Existence	0	0	0
Task Precondition - Data Value	2	0	2
Task Postcondition - Data Existence	0	2	2
Task Postconditon - Data Value	0	0	2
Event-Based Task Trigger	2	2	2
Data-Based Task Trigger	1	2	2
Data-Based Routing	2	2	2

Table 8.2: Data pattern evaluation

Appendix B

This appendix presents the operational semantics [132] of SCaLE. The purpose of operational semantics is to describe the effect of each statement as an operation, or set of operations in an abstract machine. The effect is given for simple statements (leaves). For compound statements (nodes) a reduction rule is given to combine the effects of its underlying statements. For the purposes of simplicity, we consider that SCaLE statements consist only of assignment and sequential composition. However, as we have shown in 4.4.6, conditional statements and loops are also possible. We start by presenting the statements of SCaLE and later on we move on to describe their semantics.

S ::=

skip — no-operation statement

var v — variable definition. The symbol v is used to denote a state variable. Since data types are are supported var v:T is also acceptable where T denotes a data type.

v:=val — assignment

S1;S2 — statements in sequential composition

create A a — action creation

effect E e — action effect E

input I i — action input I

start A a — starts action a

stop A a — stops action a

The approach we use to describe the semantics of SCaLE is similar to the one coined by Van Roy et. al in describing the semantics of the Oz programming language [168] and other concurrent programming languages such as KOMPICS [9]. A reduction rule of the form:

$$\begin{array}{c||c} \text{A} & \text{A'} \\ \hline \text{s} & \text{s'} \end{array} \text{ if C}$$

states that "a computation can only perform a transition from a multiset of compound actions A connected to a store s, to a multiset of compound actions A' connected to a store s' if condition C holds". However, in our case we chose to go with a session based data store and therefore in our transitions actions take place within session s and later on in session s'. Assuming there is a function id, providing the identity of each session, then for all reduction rules id(s) = id(s'). Each session is stored in a data store σ and as such it used to partition each data store. A function $\sigma(s)$ yields a multiset of keyvalues (K,V) which is a collection of the different keys and values available in each session.

We have chosen this style to describe reduction rules as opposed to more conventional formulations used for describing operational semantics such as: $\pi\langle v := val\rangle \rightarrow [v \mapsto \sigma(v)]$ (used to express assignment) or Petri nets since such formulations may grow unreasonably in length and make it harder for the reader to keep track of the session based nature of our system. Moreover, we will only provide semantic descriptions for the statements of assignment, effect and input only and omit sequential composition, the empty statement, create, start and stop since, at least on a semantical level, we are improving upon the Oz language by adding an optional type system via the use of function infertype().

Assignment

$$\begin{array}{c||c} \pi\langle v := val\rangle & \pi\langle skip\rangle \\ \hline \text{s} & s \wedge \pi(v) = val \end{array} \text{ if } s \models v \in V_\pi$$

If the variable has been annotated with it's type, if $\exists infertype(val, s) \wedge \exists infertype(v, s) \rightarrow s \models v \in V_\pi \wedge infertype(v) \subseteq infertype(val, s)$

This rule reduces an assignment to an empty statement if v belongs to the set of values of session s and its type (if there is such) is assignable in relation to the type of val.

Effect

$$\begin{array}{c||c} \pi\langle\text{effect E e}\rangle & \pi\langle skip\rangle \\ \hline \text{s} & s \wedge s' \end{array} \text{ if } s \models e \notin E'_\pi$$

To handle the optional types, if $\exists infertype(e,s) \rightarrow s \models e \notin E'_\pi \wedge E = infertype(e,s)$

Where $s' \equiv e' \in E'_\pi \wedge e \in E'_\pi \in A_\pi \wedge e \in infertype(e,s)$

Effect e of type infertype(e,s), defined by variable introduction, is added to context π. This makes the effect e visible in π and the effect e' in π'.

Input

$$\begin{array}{c||c} \pi\langle\text{input I i}\rangle & \pi\langle skip\rangle \\ \hline \text{s} & s \wedge s' \end{array} \text{ if } s \models i \notin I'_\pi$$

To handle the optional types, if $\exists infertype(i,s) \rightarrow s \models i \notin I'_\pi \wedge I = infertype(i,s)$

Where $s' \equiv i \in I_\pi \wedge i' \in I'_\pi \in A_\pi \wedge i \in infertype(i,s)$

The reduction rule for input is defined in symmetry to the effect reduction rule. Input i of type infertype(i,s), defined by variable introduction, is added to π. This makes input i visible in π and the input i' in π'.

Deadlock detection

This appendix section discusses how state of the art algorithms for deadlock detection can be applied on detecting deadlocks in application skeletons defined by SCaLE. As we have shown in 5.2.2 application skeletons in SCaLE are stored as directed graphs in a graph database and as such, any deadlock detection algorithm that runs on graphs can be used in order to detect cycles in the graph and therefore *possible* deadlocks. We emphasize here the word possible, since a cycle in a graph does not necessarily denote a deadlock, but rather the possibility of a deadlock, once the application skeleton is instantiated at runtime, assuming that due to timing reasons, certain dependencies of actions are not met.

The process for identifying deadlocks is described in [158]. In order to figure out a deadlock, an application skeleton must first be converted into a resource allocation graph. A resource allocation graph is a directed graph that consists of a set of vertices V and a set of edges E.

The set of vertices V is partitioned in two different types of nodes: P_is and R_jS.

- A directed edge from process P_i to resource type R_j is denoted by $P_i - > R_j$; This signifies that process P_i has requested an instance of resource type R_j and is currently waiting for the resource (request edge)
- A directed edge from resource type R_j to process P_i is denoted by $Rj - > Pi$; This signifies that an instance of resource type R_j has been allocated to process P_i (assignment edge).

An arc from a resource node to a process node means that the resource has previously been request by, granted to and is currently held by that process. Given the definition of a resouce-allocation graph, it can be shown that, if that graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. In other words, the deadlock detection problem is reduced to a cycle detection problem in directed graphs and algorithms such as Tarjan's strongly connected components algorithm [160].

As we have shown in 5.2.2 an application skeleton is represented as a direct acyclic graph. To convert an application skeleton graph to a resource allocation graph, one needs to map an action into a process and a variable into a resource, preserving of course the corresponding edges. This mapping yields a resource allocation graph which later on can be used for the purposes of detecting cycles.

Curriculum Vitae

Name:	Konstantinos Vandikas
Date of birth:	1982, July, 03
Place of birth:	Kozani, Greece
Nationality:	Greek

09/1997-06/2000	3rd Lyceum of Kozani, Greece
09/2000-09/2004	Ptychion (B.Sc) in Informatics at Aristotle University of Thessaloniki, Greece
09/2004-09/2006	Master of Science (M.Sc) in Computer Science at University of Crete, Greece
09/2004-09/2007	Research assistant at Human Computer Interaction Laboratory at ICS FORTH, Heraklion, Greece
09/2007-09/2011	Research Engineer at Ericsson GmbH, Eurolab R&D, Aachen, Germany
since 09/2011	Senior Research Engineer at Ericsson AB, Kista, Sweden