Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Vladislava Larina

Unified framework for omnichannel communication services

Master's thesis in Communication Technology
Supervisor: Kornschnok Dittawit
Co-supervisor: Bjørn Gulla
June 2021

NTNU
Norwegian University of Science and Technology

Vladislava Larina

Unified framework for omnichannel communication services

Master's thesis in Communication Technology
Supervisor: Kornschnok Dittawit
Co-supervisor: Bjørn Gulla
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

NTNU
Norwegian University of
Science and Technology

Title: Unified framework for omnichannel communication services

Student: Vladislava Larina

Problem description:

The evolution of communication channels as well as accompanying customer behaviour patterns have led to the emergence of omnichannel concept. The term refers to a seamless integration of different communication channels resulting in a consistent customer experience. Approaches to omnichannel transformation have been well elaborated in the marketing domain. However, such transformations strongly rely on the underlying technologies. Communication services, provided and enabled by telecommunication operators and their technology vendors, are fundamental to omnichannel transformation technologies. Integration of communication services has been to some extent addressed in unified communications (UC) solutions existing since 2010. UC solutions primarily focus on providing a single view to an end-user rather than enabling a uniform system behaviour across the supported channels. Omnichannel communication services are called to bridge the gap. However, little respective research and traditionally employed siloed design of communication services make the technological omnichannel transformation challenging.

The objective of the thesis is to elaborate on an omnichannel-enabled conceptual framework capable of incorporating flexible business logic and to evaluate this framework's viability in terms of its potential benefits and limitations by implementing a prototype for a selected scope. The project is comprised of the following major steps:

- conducting a broad study of existing knowledge context and derive functional requirements for an omnichannel-enabled framework;
- designing conceptual framework based on the requirements identified;
- defining a scope of features / services for framework validation;
- proposing architecture for an experimental implementation of the framework designed;
- developing a prototype based on the framework design and selected scope;
- performing validation of the framework with a help of the prototype developed.

Date approved: 2021-02-08

Supervisor: Kornschnok Dittawit, NTNU / Gintel AS

Co-supervisor: Bjørn Gulla, Gintel AS

Abstract

Omnichannel represents the latest stage in evolution of communication services and places strong focus on customer-centricity and unification aspects in development of such services. Omnichannel also constitutes a prerequisite to a successful digital transformation. Traditional siloed implementations should give way to flexible and cost-effective solutions which would enable high level of business logic reuse as well as fine tuning per customer needs.

Despite the market appeal of omnichannel, there has been little to none consolidation of efforts on behalf of both research community and industry to establish a common understanding of implications of omnichannel as well as to suggest any best practices for omnichannel enablement. This thesis represents a fair attempt to study existing disperse research alongside commercial offerings and elaborate on requirements and conceptual model of a unified framework for omnichannel communication services. Prototyping is used as a means to validate the proposed model in terms of its feasibility. It also allows to get additional insights insights into the topic of the omnichannel enablement and its particular aspects. The insights are not limited to the framework proposed but offer a broader perspective on omnichannel enablement in terms of its associated challenges and prospects.

Preface

This master's thesis is my final assignment for Master of Science degree in Communication Technology at the Department of Information Security and Communication Technology at the Norwegian University of Science and Technology (NTNU). The project is conducted throughout the spring semester 2021 as an in-depth evolution of the specialization project carried out one semester earlier.

The topic of the research is suggested by Gintel AS, a technology vendor for Communication Service Providers (CSPs), and thus the research itself is heavily practice-oriented. The work is supervised by Kornschnok Dittawit and co-supervised by Bjørn Gulla, both representing Gintel AS. Kornschnok Dittawit is also the responsible professor for the project at NTNU.

Acknowledgements

One does not go far alone, and this thesis is no exception. It would not happen, if not for all the wonderful people in my life continuously encouraging, inspiring, or leading me - or all things together.

First of all, I would like express my deep appreciation to my supervisors *Bjørn* and *Peach*, whom I've also been privileged to call my colleagues, for their incredible mentorship and all the time invested in me and this project.

I would also like to thank my dear family for their unconditional love and support and for being my safe haven.

Many thanks go to my friends for all the joyful distractions they caused me: after all, all work and no play makes Jack a dull boy.

Last but not least, I am grateful to NTNU and Norway for opening their doors to me and helping me, despite all the odds, to make my life-long dream come true.

Contents

List of Figures

List of Acronyms

CSP communication service provider.

CX customer experience.

DFC Distributed Feature Composition.

DTMF dual-tone multi-frequency signaling.

IMS IP-Multimedia Subsystem.

NTNU Norwegian University of Science and Technology.

SCE Service Creation Environment.

SEE Service Execution Environment.

SIP Session Initiation Protocol.

UC unified communications.

UCC unified communications and collaboration.

VoIP Voice over Internet Protocol.

Chapter 1
Introduction

1.1 Context

Digital transformation is an established trend for communication service providers (CSPs) aimed at leveraging new technology to build all-digital brands. The ultimate goal of a digital transformation is to grow CSP's revenues through reducing operational costs, increasing efficiency and customers loyalty. Improving customer experience (CX) allows to achieve higher customer satisfaction and reduce customer churn. Omnichannel transformation is one of digital transformation initiatives addressing CX enhancement by focusing on ensuring seamless and consistent interactions between CSPs and their customers across a variety of channels [Fin20]. According to [TMF18] and [Tur15], CX becomes a key differentiator for CSPs offering otherwise similar services, so CSPs have to close the omnichannel gap to retain the market [CC17, Cox18].

Digital transformation initiatives have been ongoing since 2015, but they were not prioritized and lacked alignment [NCT+20]. The COVID-19 pandemic revealed the significance of digital transformation as rates of customer conversion to digital channels have accelerated [BKv+20, OU20]. At the same time, the proliferation of such channels drives customers' expectations of a holistic CX. Although unification of customer engagement channels through omnichannel transformation has become a strategic priority, it is projected that half of large organizations will fail to meet this objective by 2022 [ML20].

As a matter of fact, many of out-ot-the-box omnichannel solutions available on the market today are implemented as discrete silo offerings which require further integration efforts in achieving tailored CX [WSP20, McE20]. Tight coupling of channels and fixed scenarios resulting in monolithic architectures is a common phenomenon. Not only such an approach results in high customization and maintenance costs, but it also acts as a barrier to digital transformation [FMWS16, NGO+19]. Thus, digital transformation calls for a shift from monolithic architectures towards layered stacks

of modular, composable, and reusable services. The first step in this direction would be to decouple channel-specific handling from business logic and thus break channel silos.

This project is done in collaboration with Gintel AS, a software company focusing on offering rich communication solutions to CSPs. Gintel voice solution allows to easily compose complex tailor-made business logic for voice communication processing, whereas for other types of channels (chat, SMS/MMS) similar functionality is offered as a set of fixed scenarios. Gintel would like to introduce a unified core enabling channel-agnostic Service Creation Environment (SCE) and Service Execution Environment (SEE) to respectively define and execute channel-agnostic building blocks.

For Gintel, an omnichannel adaptation is seen as an introduction of a unified core enabling SCE to make use of various building blocks to configure channel-agnostic business logic and SEE to instantiate and execute these blocks while efficiently decoupling channel-specific handling from business logic. It is crucial that such a solution provides cost-efficient means to deal with business logic variability as it is now done for voice services. The core should allow for an easy integration of new channels and ensure the common processing across all the channels integrated. The unified core would enable other omnichannel enhancements in the future such as cross-channel context sharing. The reason for such transformation is twofold: a single unified solution would be more cost-efficient to maintain and extend and at the same time it would be an eligible candidate for integration into a full-scale omnichannel solution of a CSP.

1.2 Motivation and research focus

Although the omnichannel concept has existed for almost a decade [HB10], it has been mostly elaborated and extensively used in the domains of marketing and management. Technological omnichannel initiatives proved to be challenging and impeded by the ever-changing scope comprised of emerging communication channels and new customer behaviour patterns. Omnichannel concepts have been a subject to misinterpretation and speculation resulting in multichannel solutions marketed as out-of-the box omnichannel products. There has been little to none consolidation of efforts of research community and industry to create a consistent knowledge context, establish best practices, or introduce standardization when it comes to technological omnichannel transformation.

As discussed in Section 1.1, omnichannel transformation remains one of the top strategic priorities for CSPs as it, among other initiatives, constitutes a prerequisite of a successful digital transformation. Moving away from an ad hoc design and silos

implementation to a common platform will ensure a uniform operation across all the channels thus making system's behaviour more predictable, optimizing customization and maintenance costs as well as costs of system's enhancements with support of new communication channels [FLB+17]. Therefore the objective of this project is to look into the traditional approach to building communication services as well disperse findings and advancements in respect to omnichannel transformation and to elaborate on the common description of omnichannel-enabled framework concept and its successive mapping on a prototype architecture, with Gintel voice solution serving as a baseline. Although omnichannel transformation of communication services is a complex topic, the project is going to be focused on the functional aspects of building an omnichannel-enabled framework. Respective research questions were formulated during the course of the preliminary specialization project, see [Lar20], as following:

Q1: What could be functional requirements for an omnichannel-enabled framework?

Q2: What are possible design constraints?

Q3: Is such a framework viable given its potential benefits and limitations?

1.3 Methodology

The project aims to contribute to omnichannel transformation by proposing a unified framework for omnichannel services as an improvement of existing solutions based on siloed implementations. Being an *improvement problem*, it is a subject to a *solution-oriented* methodology such as *design science* with a framework designed being an artifact, or an object of study. Design science iterates over two problem-solving activities: designing an artifact to improve a problem context and answering knowledge questions about the artifact in context [Wie14]. Proposing a framework is a design problem and a cornerstone topic of the project, whereas knowledge questions are motivated by the design problem and are formulated as research questions, see Section 1.2.

Both design and investigation are tightly connected, but for this project the design is primary to the investigation. Therefore, a *design cycle* as defined in [Wie14] is followed, see Figure 1.1. This cycle is a part of a larger engineering cycle which also includes implementation (application of the treatment in the real context) and its evaluation. As a *design science research project*, this project's scope is confined to the design cycle only.

Problem investigation stage should result in understanding of the context and triggers of a change and formulation of requirements through the study of existing *knowledge context*. This context is obtained with a help of *literature study* including academical and technical papers as well as professional books and whitepapers

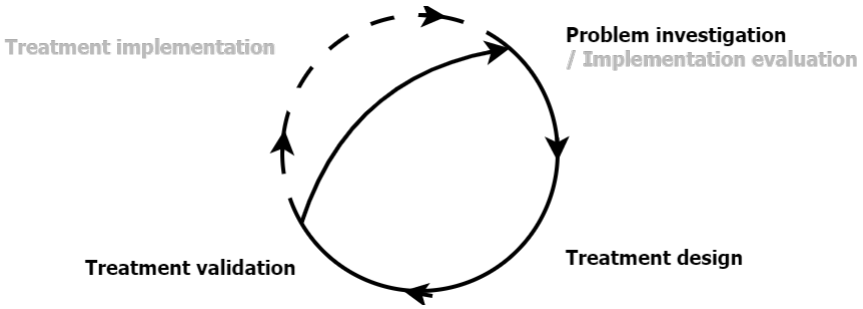


Figure 1.1: Design cycle. Adapted from [Wie14]

addressing design of communication services, including omni- and multichannel solutions as well as UC. In many cases, available specifications are defined on a high-level so that a method of *inference* has to be used in order to draw conclusions about technical details and motivation behind particular design choices. In addition to that, *technical evaluation* of the Gintel solution is going to be performed as Gintel solution is going to serve as a baseline for further work. Last but not least, the knowledge context is going to be contributed with domain insights obtained from Gintel experts during *technical interviews*. The initial problem investigation is to a great extent covered in this report.

During the treatment design stage, a proposal of a unified framework is going to be made. Here a *case-based inference* is going to be applied to a pre-defined scope based on selection of cases and channels. This method would allow to elaborate on general use framework through iterative extension of the scope during *scaling-up*. The latter is a subject to further iterations of the design cycle and hence out of the current project scope. The work is going to be occasionally supported with the same methods applied during the previous stage.

During the validation stage, an *experimental implementation*, or *prototype*, is going to be prepared. The prototype's purpose is to prove the feasibility of the architecture proposed in the previous stage. The prototype is going to be assessed in respect to its functional compatibility to the baseline, answering requirements formulated during the investigation stage and potential to further extension during scaling-up. The results of validation in their turn serve as a knowledge base for the next iteration of the design cycle. However, due to the constrained timelines and resources, the cycle is going to be limited to one or few iterations.

1.4 Thesis outline

The thesis is structured as following:

- Chapter 1 presents the context and the motivation of the project. The research questions and methodology are also defined in this chapter.
- Chapters 2 through 4 are used to establish knowledge context essential for understanding implications of omnichannel enablement when applied in the domain of communication services.
- Chapter 5 addresses the first research question by elaborating on the requirements for the unified framework as a tool to both design and execute omnichannel communication services.
- The second research question is addressed by Chapter 6 where a conceptual model of a framework is proposed and initial analysis of associated design constraints is conducted.
- Chapters 7 to 8 elaborate on the third research question by the means of prototyping and discussing its results in terms of additional insights both for the proposed framework and in a broader perspective.
- Finally, Chapter 9 provides a brief summary of the project.

Chapter 2
Omnichannel

2.1 Definition and capabilities

Omnichannel (alternatively spelled as *omni channel* and *omni-channel*) is a capability to create seamless, consistent and personalized CX across a variety of channels and thus enable customer-centricity. This capability addresses the traditional configuration of siloed channels operating in parallel rather than in concert. The concept originated and was consistently developed in marketing and management domains, but technology plays an instrumental role in the omnichannel enablement [Tur15]. The omnichannel capability is considered to be a highly demanded emerging capability of communication platforms [OU20]. This project addresses omnichannel enablement as applied in the technology domain.

Emergence of new communication channels is an ongoing process which has particularly accelerated during the past decade. In addition to traditional Voice & Telecom services (calls, IVR, VOIP etc.) and email, customers are now presented with a variety of applications and social media they can use to interact with CSPs. One of the omnichannel imperatives is to support customer's choice of channel [TMF18].

In its turn, the adoption of new channels influences customer behaviour patterns and makes customer interactions more intense and varied [CC17]. The straightforward consequence of enabling customer interaction over multitude of channels is that of customers switching between those channels during a single interaction with a CSP. Statistics gathered by [HB10] illustrate this phenomenon: 74% customers use three and more channels to complete a transaction with their CSP. Such customer behaviour pattern is called *channel hopping*: a customer moves between channels and expects to proceed with the same transaction after having moved to a new channel [Tur15, Tur17]. As a degenerate case, a customer might even use several channels simultaneously and still expect it to be handled as a single interaction. CSPs have to recognize such channel-spanning interactions and provide seamless handover by the means of preserving and passing context and data associated with the interaction.

Channel hopping support is also relevant for providing customer service automation (chat- and voice-bots, virtual assistants etc.). The employment of non-assisted channels allows CSPs to reduce human involvement and keep transaction costs low: it is estimated that between 20% and 40% of interactions volume can be handled by self-service functionalities [MRE+21, PPM18]. However, a complex situation might require a resolution in a semi-assisted manner which implies switching to one of the assisted channels. In this case, it is a must to provide smooth and transparent to the customer handover [Tur17].

Functional capabilities of omnichannel, as stated by [TMF18], address channel hopping support by the means of consistent customer's identification and application of business rules as well as data integration. CSP need to recognize the actual customer and associate it with other access identifiers across all channels and respective access rights. Customer's data should be integrated across channels, as opposed to data fragmentation inherent to traditional siloed solutions, to establish a common view of the customer [PPM18]. Omnichannel implies a high degree of automation, so it is necessary to ensure a consistent set of business rules applied to each channel to enable service consolidation and uniform system behaviour [FMWS16].

Customer data usage can be also used for gaining deeper insights into customer behaviour to drive improved outcomes. Hence, knowledge management represents another important functional capability of omnichannel [TMF18, PPM18]. Knowledge management relies on data harvesting with the successive feeding of the data gathered into analytical functions and employing the outputs to optimize operation of a CSP. For example, predictive analytics, along with Artificial Intelligence and its most popular implementation Machine Learning, are used to provide non-assisted interactions, for example, by performing tasks of automated topic detection and categorization and thus improving first call resolution rates [Tur17].

As omnichannel enablement relies heavily on customer data processing, it has to adhere in such processing to existing legal requirements such as General Data Protection Regulation (GDPR), applied to organizations operating within EU. This has a direct impact on user identification procedure as well as on personal data transfer, storage, and synchronization which should in no way compromise customer's security and privacy [PPM18, PMP18]. Privacy capability of omnichannel is critical to retaining customer's loyalty and trust as essential components of CX [TMF18].

2.2 Maturity model

Omnichannel maturity model, suggested by [TMF18], allows CSPs to evaluate their maturity in respect to omnichannel enablement. Maturity levels represent evolution of omnihchannel, see Figure 2.1, with its predecessors defined as:

– *Single channel*: disparate channels for customer interaction and siloed operation on CSP's. Each channel offers a unique CX.

– *Multi-channel*: CX across channels has increased consistency but remains fragmented. Customers are restricted to one channel during a transaction.

– *Cross-channel*: provides initial channel integration capabilities and consistent view from a customer's perspective.

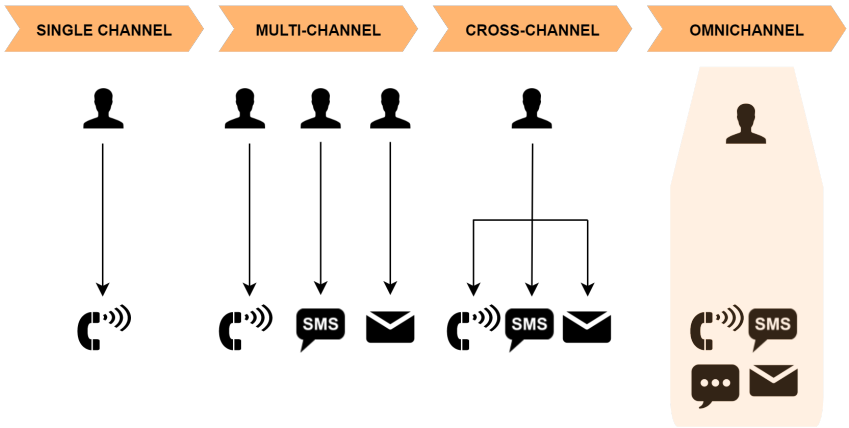


Figure 2.1: Omnichannel evolution. Adapted from [Tur15]

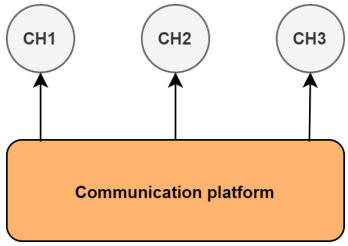
The model defines maturity across multiple, not exclusively technical, dimensions. A CSP may have different level of maturity in different dimensions.

It is worth noticing, that the terminology presented above is not uniformly adopted in the industry which reflects absence of common understanding. In addition to that, there is a marketing trend to attribute omnichannel properties to the products which are in their essence multi- or cross-channel [WSP20]. That leads to the general depreciation of the terminology as such.

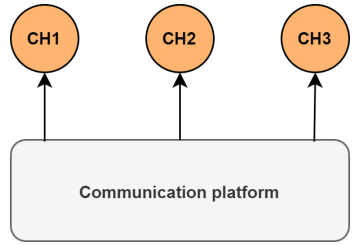
2.3 Approaches to omnichannel enablement

Being one of the main drivers of the structured approach to the omnichannel enablement, TM Forum have identified and assessed a number of approaches to omnichannel enablement in [TMF18]. The main approaches are demonstrated on Figure 2.2 with parts of a system which are subject to change colored with orange.

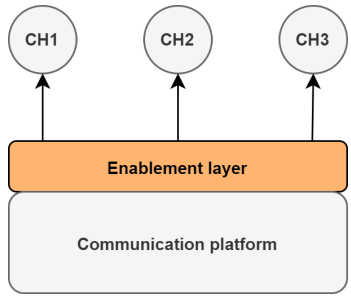
The approaches vary greatly in terms of associated risks, costs and functional potential. For example, platform-based enablement (see Figure 2.2a) exists in two variations: a greenfield approach (replacement of the current communication platform)



(a) Platform-based enablement



(b) Channel-based enablement



(c) Encapsulated enablement

Figure 2.2: Approaches to omnichannel realization. Adapted from [TMF18]

and an incremental transformation of the current communication platform. The greenfield approach allows to address the complete omnichannel feature set with a promise of full interoperability as no alignment with legacy systems is required. However, such a solution comes with particularly high risks and costs, in particular, capital expenditures (CAPEX). The incremental approach introduces changes in a more controllable way and allows to retain legacy processes. On a downside, it comes with a risk of breaking such legacy processes. Moreover, the incremental transformation inherits limitations of the the current platform thus restricting a full-fledged omnichannel enablement.

Channel-based enablement (see Figure 2.2b) could be considered the most safe and economical way to omnichannel enablement, had it not been highly constrained by the existing core stack. This approach is not likely to lead to the required level of consistency between channels.

Finally, there is an option to encapsulate omnichannel capabilities in an enablement layer, see Figure 2.2c. Much like the incremental transformation, such an approach is influenced by limitations of the current platform, but allows to orchestrate implementation to introduce system complexity in a more controllable and

agile manner.

2.4 Example of industrial implementation

The concepts discussed in Section 2.1 can be illustrated with an example of industrial implementation. Genesys Engage is as a full-featured omnichannel engagement solution comprised of a broad range of products [Gend]. It is also an only found example of a commercial solution with publicly available technical specification defined at a proper level of detail and from a perspective relevant to the discussion of omnichannel enablement.

From the omnichannel perspective, Conversation Manager (CM) is a cornerstone component of Genesys Engage solution, see Figure 2.3. CM enables orchestration of CX across a multitude of channels using customer engagement context and business rules to decide when and what action to take. The context awareness is addressed by Context Services (CS) sub-component of CM, and Genesys Rules System (GRS) decides on business logic to be applied. The logic is then executed by another channel-agnostic component, Orchestration (ORS).

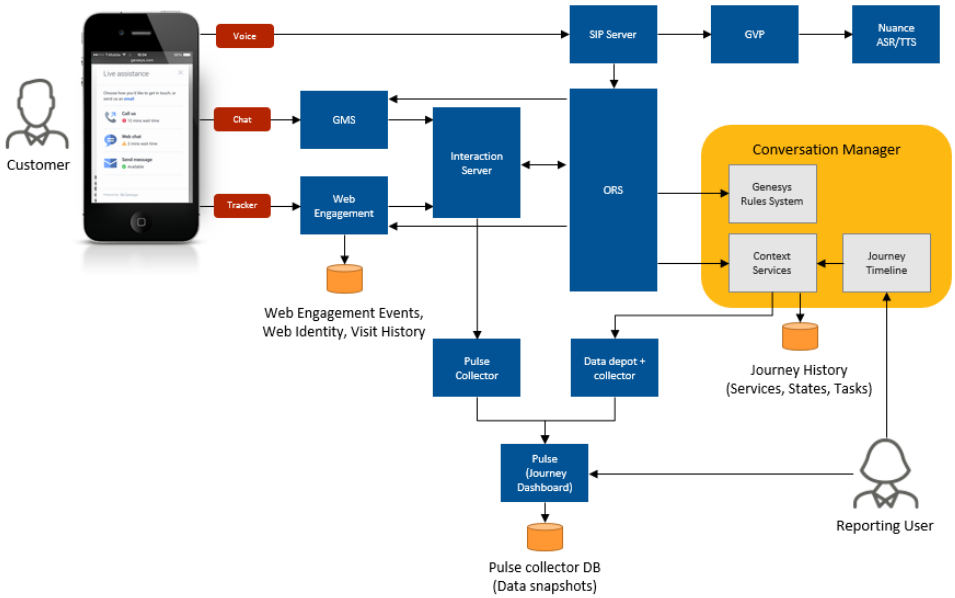


Figure 2.3: Genesys Engage high level architecture. Source: [Genb]

In addition to publishing the high level architecture of their solution, Genesys also elaborate on some concepts underlying the omnichannel implementation. For

example, an omnichannel *conversation* is defined as a composite entity comprised of an arbitrary number of interactions spanning an arbitrary number of channels during an arbitrary period of time with those interactions related by context. The *context* is further defined as a combination of a service, a state, a task, and other relevant to the conversation data [Genb]. *Service* is a business process comprised of one or more communications between a customer and an enterprise. A service can be divided into a set of *states* to transition between them. A state can in its turn include a list of *tasks* [Gena]. Such a structure of nested entities, see Figure 2.4, allows to define context with high granularity which then helps to introduce flexible business logic to handle each step of a customer's journey in an optimal way.

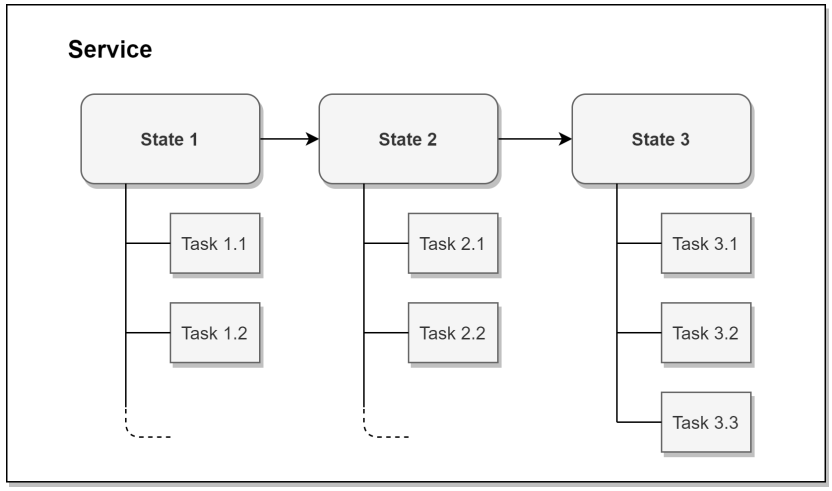


Figure 2.4: Genesys Context Services resources. Adapted from: [Gena]

The business logic is described by the means of *business rules* grouped in *rule packages*. A package contains an arbitrary number of rules and an associated *fact model* describing input for and output of a rule execution [Genc]. A business rule is then included into a *strategy* to define a workflow. Genesys provide channel-agnostic SCE and SEE to respectively define and execute business rules.

As both Genesys and Gintel place strong focus on business logic flexibility (see Section 1.1), the former makes a particularly relevant example of existing omnichannel implementation. Some of the concepts defined in Genesys Engage, for example that of a conversation, are going to be further leveraged during design of omnichannel-enabled conceptual framework.

Chapter 3
Unified communications

3.1 Motivation of study

The term *unified communications* (*UC*), also known as *unified communications and collaboration* (*UCC*), refers to a class of communication solutions integrating various communication channels (voice, video, messaging etc.) to provide a common end-user view. Although originally UC does not imply having omnichannel capabilities, some UC vendors (for example, Mitel, Vonage, Avaya OneCloud) explicitly declare omnichannel support [TFO+20, OU20]. That combination is not surprising as broader UC offerings also include an integrated contact center, and omnichannel enablement to a great extent addresses issues of a contact center [Boc17].

There are also other aspects of UC which make it particularly relevant to the discussion of omnichannel. UC address continuity of user experience in the sense of device handover: user's communication activities have to be seamlessly transferred among multiple devices running UC so that the user can switch between those devices and pick up where the user left off [PMF+17]. Device handover is enabled by maintenance and continuous synchronization of a shared context between instances of UC running on different devices. The shared context is also required to provide a unified view of various communication channels to the end user. For example, a conference call is often enhanced with a chat for the conference participants, and the common participants list have to be maintained both for the call and the chat so that users who joined or left the call are also added to or removed from the chat, respectively. Now, it is easy to draw a parallel with the context preservation during the channel hopping (see Section 2.1).

Such an overlap in the topics of omnichannel and UC allows to use the latter as a complement to the problem context and try to gain additional technical insights into the topic of communication channel merge.

3.2 Technical insights

This section makes an effective summary of technical insights gained from studied research papers on topic of UC and open documentation on UC commercial solutions.

Despite a plethora of communication channels, any communication can be effectively classified as being either asynchronous or synchronous. The latter (such as a voice or video call) can naturally be perceived as a *conversation*. A conversation relies on the availability of all the parties involved and requires complex setup, handling, and teardown with a state maintained throughout the duration of the conversation. An asynchronous communication (such as an e-mail, SMS/MMS) can be denoted as a *notification* which requires neither availability of the destination party nor maintenance of a state. Since UC aims at providing a common processing for all kinds of communication, a notification is considered to be a special case of conversation thus allowing to define a common denominator in the processing of both types of communication [HR04, GCM12]. Using Genesys Engage discussed earlier as an example, one could follow the natural evolution of such UC terminology in regards to omnichannel.

Common communication handling also implies usage of a flexible and extensible control protocol. The common choice for such a protocol is Session Initiation Protocol (SIP) [ZJ10, TLCZ14, BGK16]. SIP is an application-layer control protocol for handling multi-participant multimedia sessions. It provides primitives, or mechanisms, that can be used to implement meaningful services as well as extension points to introduce custom operations [IETa]. SIP can also be used to carry data, but normally a separate point-to-point path is established for media flow, see Figure 3.1.

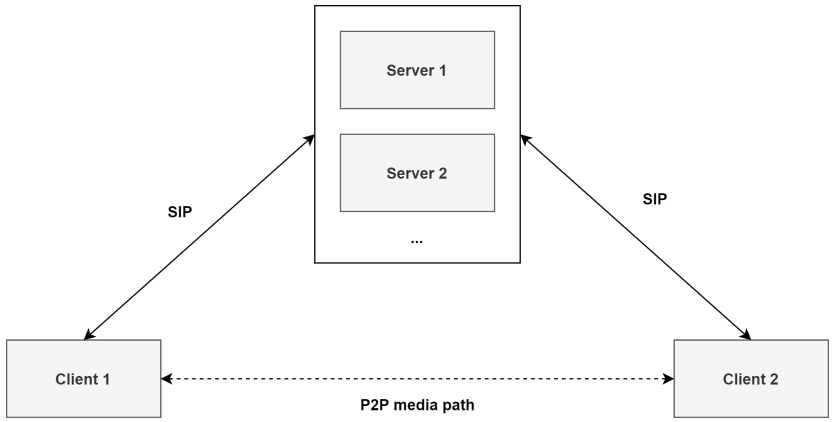


Figure 3.1: UC reference model. Adapted from: [ZJ10]

Research on UC topic touches upon the context-awareness as a requirement for

UC to be embedded with business systems [HR04, GCM12, RT09]. It is proposed to encapsulate such a functionality into a separate component, for example, a stand-alone extensible Context Service with an exposed API [HR04, HSD+02] or Contextualization module providing work context as described in Section 3.1 [RT09]. Once again, these ideas have a clear resemblance with the omnichannel-enabled architecture as discussed for Genesys Engage.

The insights discussed have been derived from the disparate research. Despite the high expectations, the study of existing offerings, both omnichannel and UC, with the exception of Genesys Engage, failed to provide any applicable inputs for the omnichannel discussion. The study included but was not limited to Omnichannel Enterprise Call Center and Contact Center by Mitel [Mit], Omnichannel Contact Center by Cisco [Cisa] and Unified Communications and Collaboration by Cisco [Cisb].

Chapter 4
Distributed Feature Composition

4.1 Main definitions

Distributed Feature Composition (DFC) is a component-based approach for specification and implementation of telecommunication services. It was first introduced in 1998 in [JZ98] and then further elaborated and used in subsequent research in design and development of media services. DFC has also been employed for implementation of both commercial Voice over Internet Protocol (VoIP) solutions, such as AT&T CallVantage [BCG+05], and numerous prototypes and demonstration services [Zav09, ZC09].

The central concept of DFC is that of *a feature* which is an increment of functionality enhancing a basic communication capability. A *service* is a set of features which is usually used to market and sell those features as a whole. In this sense, features are concepts to explain a behaviour of a service to its users [BCG+05]. From the software development perspective, a feature is a standalone component which encapsulates some common functionality and could be independently specified, developed and tested, whereas a service is a specific composition of such components [JZ98, Smi10].

DFC is a domain-specific adaptation of pipes-and-filters architectural pattern which enables structured composition and promotes principles of loose coupling and reusability. This pattern allows to design a system as an easily modifiable graph where nodes (filters) represent context-independent components linked by edges (pipes) representing data streams. In DFC, features play the role of filters and pipes are realized by the means of internal calls [ZJ02]. Each request for service is then satisfied by a dynamically assembled signalling graph of features corresponding to the services enabled for the parties involved in the communication. An example of such a signalling graph is presented on Figure 4.1 and describes setup of communication from party A to party B where the latter has Call Forwarding Busy (CFB) feature enabled. In case a busy signal is received, CFB alters the signalling graph by creating a new communication leg to party C while terminating a previously created leg to

party B. All the parties are connected to the DFC graph via interfaces (I) which are also features. The signalling path is comprised of features (F) defined in services assigned to the parties.

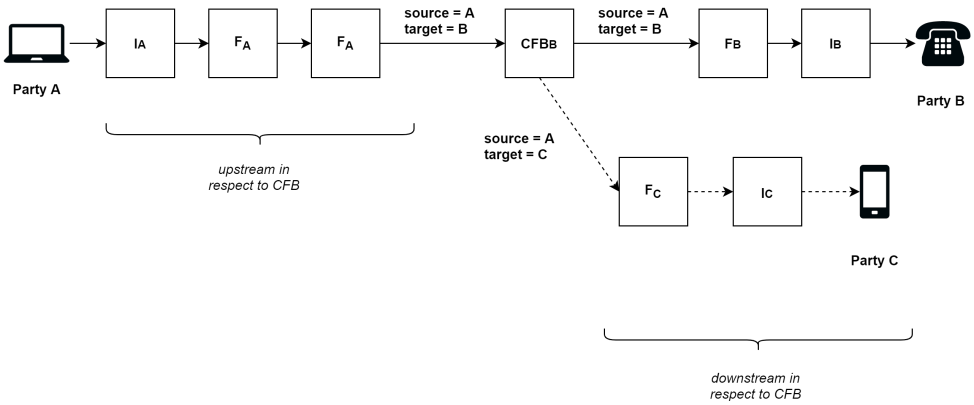


Figure 4.1: Example of DFC setup signalling graph. Adapted from: [Zav09]

In DFC, each feature should meet the following requirements [Zav09, Zav11]:

1. *transparency*, in the sense that an inactive feature merely relays signals and hence is not observable. In the example on Figure 4.1, CFB initially behaves transparently and does not alter up- or downstream signals, unless the latter conveys a busy status.

2. *autonomy*, meaning each feature can act as a protocol endpoint without reliance on other features. Sitting in the signalling path, a feature is able to observe all the signals as well as alter or absorb them or generate new ones. In the example used above, CFB is enabled to take an autonomous action of communication legs re-configuration based on analysis of a downstream signal.

3. *context-independence* as a feature neither relies on nor has any knowledge of other features involved. In terms of the used example, CFB acts on a busy signal in the same way independently of whether that signal is generated by a user device or another feature box such as, for example, Do Not Disturb (DND) feature which could be used to simulate a busy status of party.

To perform their functionality, features may rely on persistent *operational data*. This data can be manipulated both by features and through a provisioning interface separate from communication processing. Provisioning stands for introducing user and configuration data into a system. For example, provisioning of CFB feature for a

particular user involves setting a forwarded-to address (party C). This value is then retrieved by CFB instance during a communication processing.

The operation data could be used for feature cooperation as illustrated on Figure 4.2. Here, operational data is used to store buffered messages which delivery initially failed. Although operational data represents a valuable channel of communication between features, it opens the door for accidental *feature interactions* [BCP+04]. A feature interaction occurs when one feature is affected by another feature so that the behaviour of the affected feature deviates from its specification and may lead to the service disruption [CGL+93]. Since the main motivation behind DFC is to avoid feature interaction, the shared use of operational data is generally discouraged. Instead, it is recommended to partition operational data by feature and address (served user) so that a feature can only access its own data relevant to a specific service invocation [JZ98, BCP+04, Zav09].

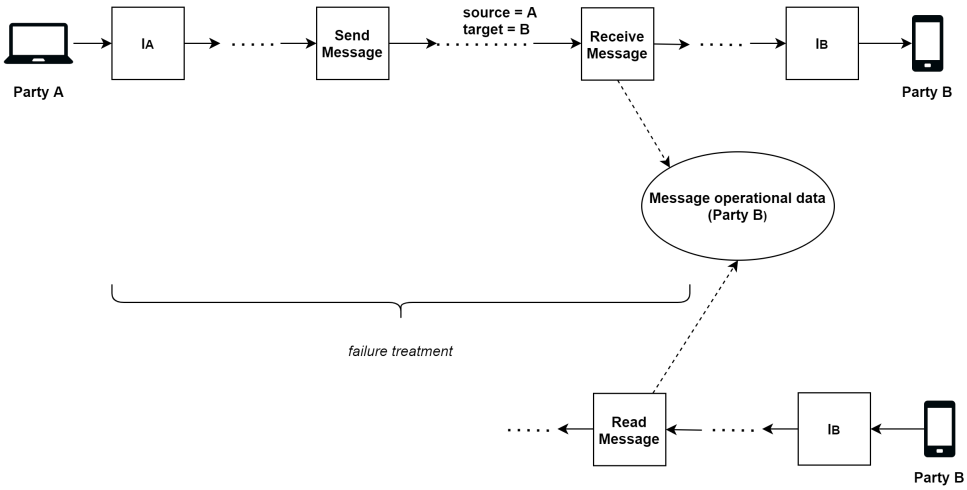


Figure 4.2: Example of shared operational data. Adapted from: [BCP+04]

Apart from the operational data, DFC distinguishes routing data such as *subscription data* indicating services assigned to a user and *precedence data* governing the relative order of features. Routing data is used by *feature router* to assemble a graph of features on communication setup [BCG+05, Zav11]. Having received a setup signal, a feature forwards request to the feature router which then calculates next feature to forwards the signal to, see Figure 4.3.

The feature router and features interact via an abstract communication control protocol, or DFC control protocol. The protocol consists of command (setup, teardown etc.) and status (available, non-available etc.) signals. Although DFC does make a proposal of such a protocol [ZJ02], it does not necessarily correspond to the

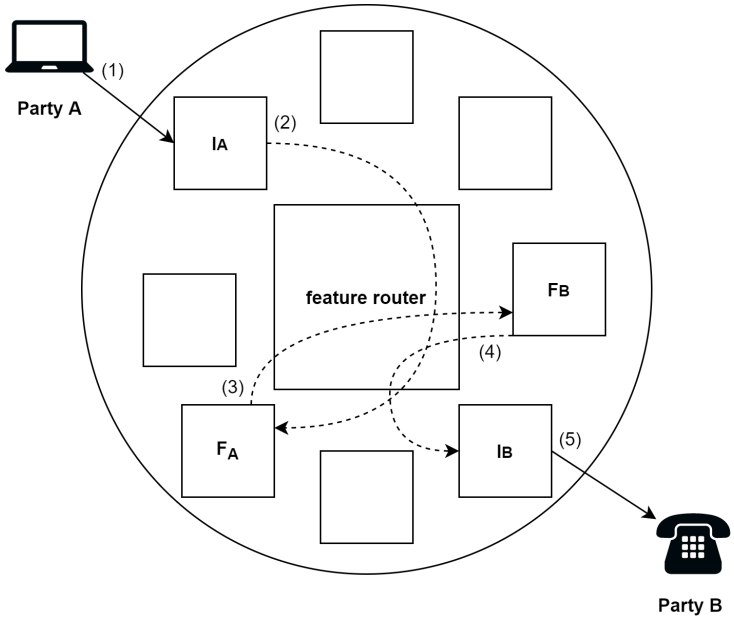


Figure 4.3: Communication setup. Adapted from: [JZ98]

required level of modularity in each particular case. More specialized signals allow to define features in a fine-grained way and build more advanced services. However, designing such signals in a featureless way is challenging and not necessarily feasible [Zav09, CS09].

4.2 Omnichannel potential

Initially, DFC did not consider having distinct paths for media and control signalling. However, a need for such a separation became apparent during DFC implementation. Firstly, these types of signalling have different requirements in respect to bandwidth, reliability, etc. Secondly, content of a media channel may be concurrently influenced by a set of features which calls for a separation of service and media control as shown on Figure 4.4. Thus, features can be stripped of media-specific logic and defined in a channel-agnostic manner [BCP+04, Zav09, ZC09].

Having channel-agnostic features and a control protocol accommodating both synchronous and asynchronous communication, one could design a unified service. For example, the service illustrated on Figure 4.2 could be extended to voice channel with "Send Message" standing for starting a call with a fallback option of recording a voicemail and "Read Message" standing for listening to the deposited voicemail [BCP+04].

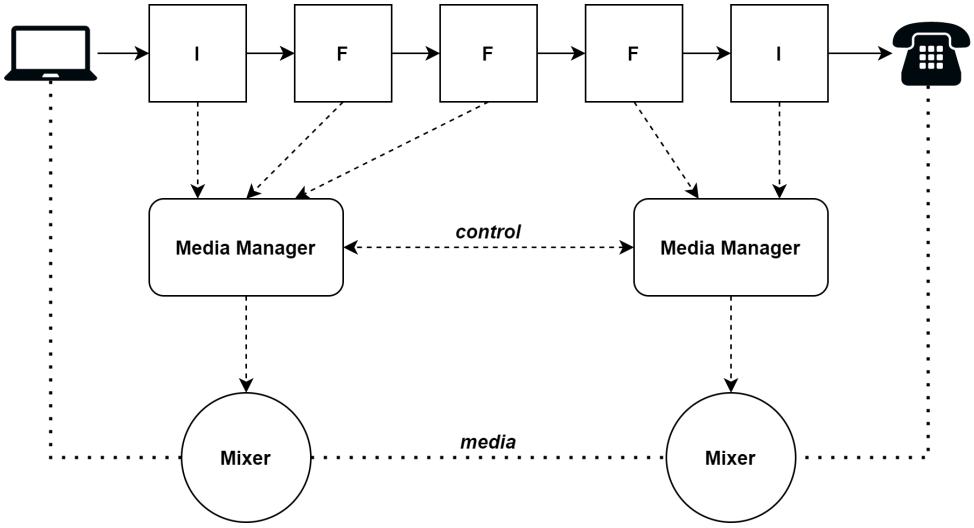


Figure 4.4: Separate media control in BoxOS implementation. Adapted from: [BCP+04]

A user invoking a service determines a communication channel by choosing an end device. End devices operate in communication protocols different from the DFC control protocol. Interoperability between an end device protocol and the DFC control protocol is enabled by dedicated interfaces, or adaptors. One could define multiple adaptors as potential entry points of the same service and thus allow that service to be invoked for the supported channels.

Moreover, [BCP+04] discusses a possibility of handling different channels in the same service simultaneously by introducing a facade component making devices supporting different channels function as one. However, that idea was not further elaborated and did not find a way into the respective implementation.

By allowing one to define a service in a channel-agnostic way, DFC becomes extremely relevant to the discussion of omnichannel, especially given the general lack of technical insights thereof. DFC serves as a suitable candidate for further discussion and design of a unified framework for omnichannel communication services.

Chapter 5
General framework requirements

The theoretical baseline discussed in the previous sections allows one to derive general requirements for a uniform framework for omnichannel communication services. In the context of the discussion, these requirements are considered functional as they address different behavioural aspects of the framework as a tool for services definition and execution. The requirements are grouped with respect to such aspects.

5.1 Modularity and service flow

On the one hand, modularity requirements are driven by best practices in telecommunication services design which promote service flexibility and feature reusability. On the other hand, modularity requirements are suggested by the very definition of omnichannel. An omnichannel service shall support channel hopping which implies a possibility of a service interruption with a successive service resume. That notion suggests a non-monolithic structure of a service.

Hence, the framework shall define a service as a composition of independent blocks of functionality, or features. A service defines one or many *service execution paths* as sequences of successively invoked features which selection is dynamically defined by the service logic. Each feature must be defined in a self-contained manner, without making any assumptions about other features or their relative ordering along service execution paths.

The framework shall define at least one default entry point for each service. An entry point is a feature triggered upon a new *service invocation*. Each service invocation shall be unambiguously identified by the service triggered and *the served user*. The framework shall define at least one condition which allows to indicate a successfully *completed* service invocation. Any alternative outcome of a service invocation shall be treated as an *interrupted* service invocation. The framework shall detect an implicit service interruption caused by channel hopping. A service invocation which neither has been completed or interrupted is considered to be active,

and the framework shall ensure all the active service invocations at any instance of time are uniquely identified by a service and a served user.

The framework may allow *service resume* as a treatment for an interrupted service invocation by defining *restore points* for the service. A restore point is a new entry point only employed for a new service invocation associated with an existing interrupted service invocation, with an association performed on behalf of the framework. A restore point shall optimize the new service invocation by effectively reducing the respective service execution path.

The framework shall allow to define *utility services* addressing cross-cutting concerns such as logging, monitoring, audit, error handling etc. The framework shall provide means for interaction of utility services with feature-based services.

5.2 Channel agnosticism

The cornerstone of omnichannel is that of providing a consistent processing via different communication channels. Hence, requirements to the framework should tackle decoupling of service logic and channel-specific processing.

Despite the abundance of communication channels, all of them could be classified as being either synchronous or asynchronous. The framework shall provide support for both modes of communication. The framework shall also support a change of communication mode during service resume thus effectively enabling channel hopping. This also implies the framework shall recognize the same user across different channels.

The framework shall define services and features in a channel-agnostic manner in respect to both business logic, data, and semantics. The framework shall define an abstract control protocol for feature interaction, or *service orchestration*. The framework shall enable conversion of external channel-specific signalling into the internal protocol and vice versa by the means of peripheral *adaptor* features. The framework shall enable media control separate from business logic expressed by features.

5.3 Data model

The framework shall define a common data model to describe users, services and service invocation states regardless of a channel in use as well as a data normalization mechanism to transform non-compliant data to the common data model model. The framework shall allow to establish *a service execution context* as a combination of global configuration data, user data and state data relevant to a particular a service invocation.

Chapter 6
Conceptual framework

6.1 Proposed model

The unified framework for omnichannel communication services can be defined in terms of classes of entities and character of their relationship. The conceptual framework proposed is based on the requirements discussed in Section 5 and the problem context study conducted during the course of the project. Speaking of the problem context study, design of the model is heavily based on Genesys Engage discussed in Section 2.4 and DFC discussed Chapter 4. The conceptual model is illustrated on Figure 6.1.

The cornerstone idea of the framework is that of clear separation between two following concepts:

- *Conversation* as an entity consolidating multiple attempts of a particular *User* to trigger execution of a specific *Service* until that execution has been successfully completed or permanently invalidated (for example, due to some expiration conditions).

- *Interaction* as one of such attempts. An Interaction has an associated *Execution State*.

In line with the previously established requirements, all currently executed Conversations are uniquely identified by a pair of a User and a Service. At the same time, a User can have multiple ongoing Conversations as long as they involve distinct Services. The illustration of the model (see Figure 6.1) does not capture a unique association between a User/Service pair and a Conversation. The reason for that is that the constraint has only an immediate effect and is not applicable in time perspective. In other words, the history of Conversations may contain entities with duplicated User-Service pair. However, this fact does not impede a differentiation of

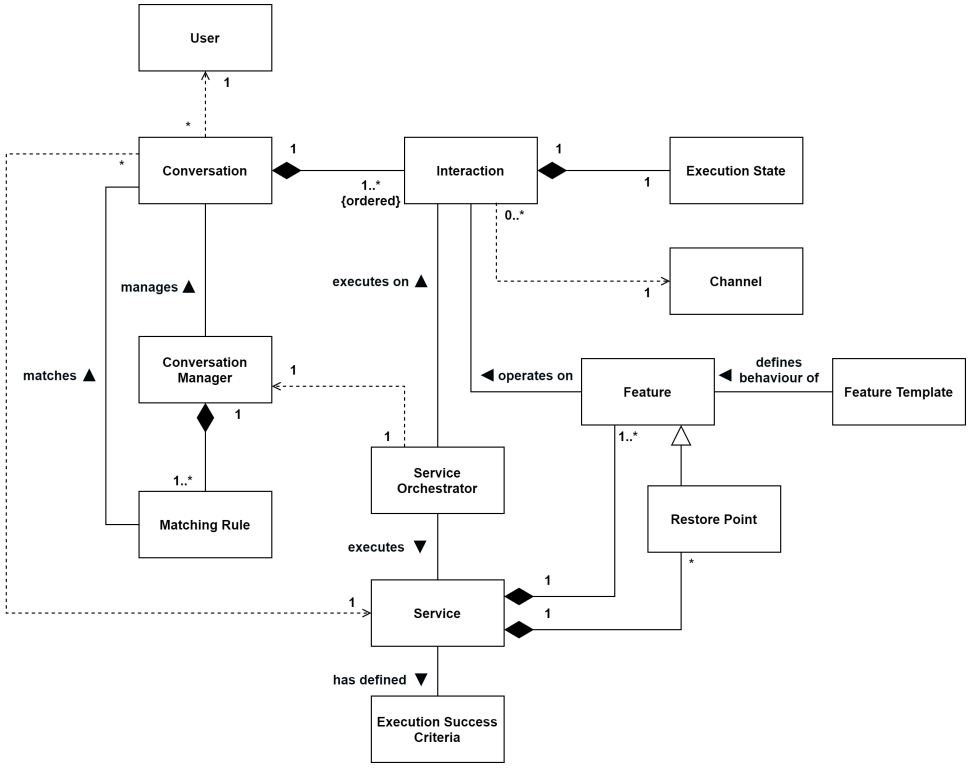


Figure 6.1: Conceptual framework for omnichannel communication services

distinct Conversations as it is possible to distinguish them based on their execution time.

The logical separation of the two entities supports notions of the continuous service execution and the service resume. If an Interaction is interrupted, it is possible to derive a non-default State for a newly issued Interaction in the scope of the same Conversation and continue with the service execution in an optimized way, without having to start the Service from scratch.

A Conversation maintains order of the associated Interactions with respect to their start time with newer Interactions normally having precedence over the older ones. Having such an order in place helps to correctly derive State upon the service resume: for example, if there have been multiple interrupted Interactions, only the most recent is used for the state derivation of a successive Interaction. The particular service resume procedure, however, is dictated by underlying business logic and can override the default order of precedence for Interactions.

Each Interaction is associated with a particular *Channel* from those offered to Users. A Conversation itself is omnichannel as it can be conducted via multiple channels without having direct dependencies on those channels. In other words, channel hopping is supported during the course of one Conversation.

The framework relies on two global entities acting in tandem to provide Users with consistent Services:

- *Conversation Manager* performing matching of Conversation and Interaction based on a set of *Matching Rules* as well as consistent management of those entities;
- *Service Orchestrator* performing feature routing based on User and Service configuration and a state of the matched Interaction.

The initial matching of pending Conversations by User and Service is performed by an implicit Matching Rule whereas explicit Matching Rules allow to apply additional business logic to narrow the selection even further. For example, an initially matched pending Conversation may be considered invalid if the last associated Interaction has a one-week-old timestamp. In this case, Conversation Manager has to permanently invalidate it and create a new one. Once Conversation matching is done, an Interaction matching in scope of this Conversion is performed: in case of a non-interrupted service execution, the currently pending Interaction is used in further processing, otherwise a new Interaction is created. Then, the Service Orchestrator uses the matched Interaction in service execution.

There are a few more concepts involved which are used to describe the business logic applied to an Interaction by the Service Orchestrator:

- *Feature Template* defining business logic unified by a single purpose as well as parametrization options of that logic;
- *Feature* as an instance of a Feature Template which parameters are set to either explicit or default values;
- *Service* as a composition of Features;
- *Restore Point* as a special sub-type of a Feature which declares that Feature is a part of the service resume strategy defined in the scope of the respective Service.

A Feature Template defines a general behaviour whereas a Feature allows to supply some specific parameters to this behavior and thus derive its service specific

version. For example, a Feature Template for Call Forwarding Feature may allow to specify a forwarded-to number in different ways (as a static value, a database query etc.) whereas a particular Feature chooses one of these options to provide an unambiguous behaviour. The separation of a Feature Template and a Feature serves multiple purposes such as reusability (as the same Feature Template can be used for setup of multiple Features both within the scope of a single Service and across a number of Services) and flexibility (as it is possible to derive variations of general behavior defined by a Feature Template by setting up Features with different parameters).

A Service may or may not have Restore Points defined which indicates this Service being resumable or not, respectively. A Service, however, must have an *Execution Success Criteria* defined to be able to recognize service logical completion and distinguish it from the situation when the Service execution was interrupted. Once an Execution Success Criteria is satisfied, the running Interaction and the respective Conversations are considered to be completed, even though the Service might still have some post-processing logic being executed. Both an Execution Success Criteria and Restore Points are defined by business rules behind the respective Service.

The Service Orchestrator executes Features on the matched Interaction which means that Features operate on Interaction effectively changing its State and delegating it any channel-specific logic. On practice, that implies Interaction providing a channel-agnostic interface supporting general operations to be performed on communication.

6.2 Design constraints

Although the conceptual model is defined on the high level of abstraction, it still could be analyzed in respect to constraints this model potentially introduces to the design of omnichannel communication services.

The first constraint relates to the notion of the conversation matching. The proposed requirement is that of limiting the number of concurrent Conversations for the same User-Service pair to one. Strictly speaking, this is not an explicit requirement of omnichannel but rather a working assumption: even if a User makes simultaneous attempts to trigger the same Service, it might be practical for the service provider to optimize the service fulfillment by ignoring or suppressing all but one such an attempt. However, specific business rules behind the service may still allow a User to have multiple parallel Conversations being executed for the same Service. Since Conversations are channel-less, such a situation may present an issue for correct matching of a Conversation due to the lack of uniqueness in the Conversation definition. As the service resume procedure relies on the conversation

matching, it also becomes non-deterministic. The added flexibility to the framework should be compensated with additional criteria for unambiguous differentiation of Conversations allowing to effectively resolve the conversation matching issue.

Another constraint of the model relates to the interoperation of Features and Interactions. As it was mentioned before, such an interoperation is enabled by a channel-agnostic interface which allows to decouple business logic defined as behavior of a Feature and channel-specific logic defined within a respective Channel entity. The model assumes that such an interface could be defined in the first place. However, the task of such an interface specification is non-trivial and depends on such factors as set of channels and operations to be supported as well as a required level of detail in an operation's definition. For example, a Termination feature defined in the Gintel solution is responsible for establishing and handling connection between two parties. This feature is defined in the terms of a voice channel and as such provides options for customized handling of a disconnect event, also distinguishing which party initiated that disconnect. However, extending this notion to an asynchronous channel is at the very least not straightforward, if feasible at all. In any case, it requires a service designer to elaborate on a well-thought semantics behind the services which could be generalized across different in their nature channels.

Chapter 7
Prototype

According to the discussed methodology, the validation of the proposed conceptual framework is done by implementing a prototype using Gintel Call Handling solution as a baseline. The current Gintel solution allows to define and execute complex tailor-made services for the voice channel only, see Section 1.1 for more details. The initial ambition is to perform the platform's omnichannel enablement for a limited scope of features and thus create a prototype of the unified framework for the omnichannel communication services. Creation of the prototype drives the empirical validation of the proposed framework by investigating the framework's feasibility, potential benefits and limitations and thus addresses the respective research question.

It is important to note that the method of validation is inherent to the topic of the research as the latter is heavily practice-oriented. However, such a method comes with certain limitations since the results of validation are strongly defined by its prerequisites (such as baseline, selected scope, etc.) and therefore might be challenging to generalize. Nevertheless, having the functioning baseline allows one to concentrate on omnichannel enablement issues and optimize time spent on feature design and protocol-related development. Results provided are expected to be sufficient for the first approximation of the unified framework design.

7.1 Example service

Although the Gintel Call Handling is not immediately based on DFC, a certain resemblance takes place, and such resemblance is sufficient to use the Gintel solution as a baseline for prototyping the proposed framework which in its term employs some of the DFC principles.

Like DFC, the Gintel solution is based on the idea of modularity where a complex behaviour is designed by composition of building blocks called components. The granularity of those components differs greatly: some are used as a mere means to conditionally branch execution paths, some could be straightforwardly mapped onto

a DFC feature like, for example, communication barring or making a prompt, and some components incorporate functionality of several DFC features. A Termination component could serve as an example of such compound functionality which not only represents an endpoint connecting the served user to a configured destination but also encapsulates various forwarding options as well as call control functionality. For the prototyping, this conceptual misalignment with a Gintel component and a DFC-like feature could be ignored, and Gintel components are treated as features.

In general, all the features can be divided into two groups: those involving channel-specific logic and those which are channel-neutral. For the prototyping purpose, a few features from both categories have been picked up to compose the example service:

- Menu which first presents the served user with an informative message and then collects user's input to determine further service execution path. This is a channel-specific feature which employs a media server for the playback of the announcement and enables user input via dual-tone multi-frequency signaling (DTMF);
- Time Router which checks if current time falls within a configured period to determine further execution path (channel-neutral);
- Termination which serves as an endpoint connecting the served user to a configured destination (channel-specific);
- Release which releases communication with a configured error code (channel-specific).

The example service has been designed with EasyDesigner (a Gintel proprietary SCE), see Figure 7.1. The example service follows the idea of a main number service. Main number services allow to streamline incoming communication processing by using the main number as an exposed contact point of a company and applying some semi- or fully automated logic to distribute the communication across company's agents. It is important to note, that the example service discussed in this section is not a real-life example and is only used for the prototype purposes. A few simplifications are made which might not be meaningful in terms of a real service.

The semantics of the example service is that of connecting a user with an agent, or destination, where the choice of the latter is conditioned by both user input and time conditions, or schedules. The possible outputs of the service are that of the served user connected to one of the four possible destinations (200, 300, 401, 402) or disconnected with an error. The latter represents one of the simplifications made for the service which might not be applicable to a real-life service. In the example

service, the Release feature is used to interrupt the ongoing interaction and test more variations of the service resume. However, a real service is unlikely to have an execution path deliberately leading to an error.

The prototype scope includes omnichannel enablement for the features used in the example service to support a message channel along with the voice one which is already being supported by the Gintel system. For the sake of simplification, these channels merely represent different communication modes with the message and the voice channels used for asynchronous and synchronous communication respectively. In terms of the underlying protocols, both channels rely on SIP with the message channel taking use of the SIP method of the same name (MESSAGE), see [IETb]. The semantics of the service as applied to the message channel is interpreted as relaying contents of the original user request used to trigger the service execution to a destination determined by the selected execution path. This is yet another simplification made for the example service as pure relay might be of a limited use for a real-life service.

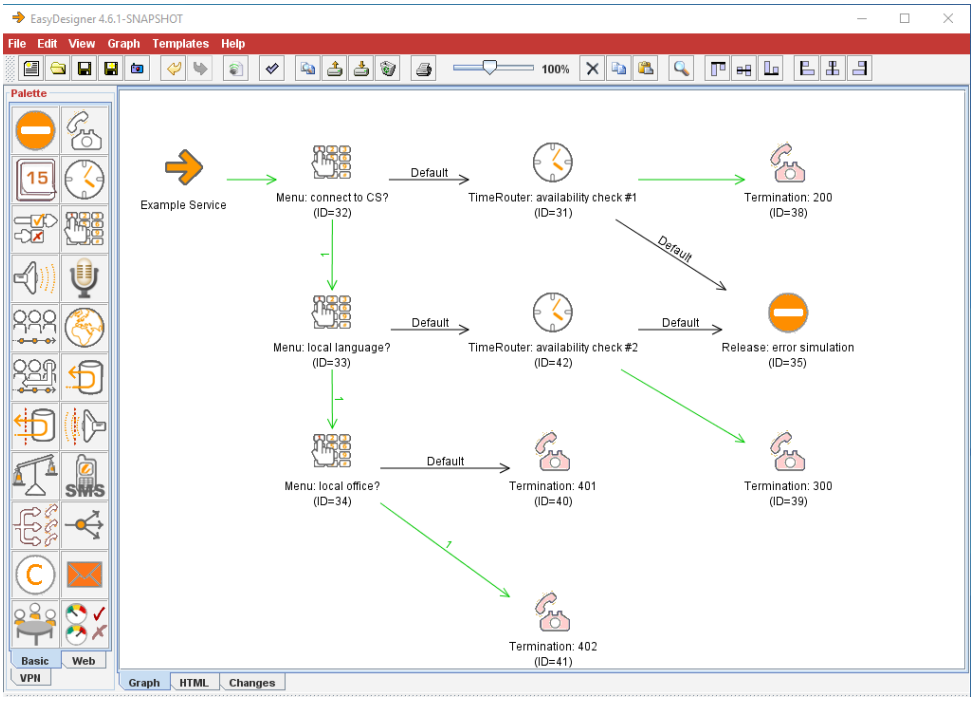


Figure 7.1: Example service created in EasyDesigner

The flow diagram on Figure 7.2 extends the description of the example service. The diagram covers yet another aspect of the business logic which has to do with

the service resume. Instead of the feature-based view, the service logic is represented by the sequence of steps. Each Menu feature is mapped onto three steps such as making a prompt, processing user input and a condition check. Each Time Route feature is mapped onto two steps such as perform a schedule check and a condition check. Both Termination and Release features are mapped onto one step each which are also the terminal nodes in the graph.

Both a default entry point and a service execution success condition are indicated along with the interruption condition explicitly introduced to the service logic. The business rules were defined in a way to consider each prompt and each connect step as a restore point for the service (colored with yellow). The diagram reveals why the Release feature was used to introduce an interruption: it is a terminal node which is nevertheless not used in the service resume and allows to verify that the service resume is done correctly in respect to the execution path which led to the erroneous step in the first place. The service execution success condition is for the served user to be connected to one of a destinations. It translates to a successful INVITE and MESSAGE transactions for the voice and message channels respectively.

7.2 Basic test scenario

The example service can be executed in a number of ways. For the validation purposes, it is practical to limit the prototype test scope to a few test scenarios. The prototype is created via an iterative omnichannel enablement of the baseline. The first iteration consists of enhancing the baseline with a support of the message channel. In other words, all the features employed in the example as well as the service orchestration should support service execution for both the voice channel and the message channel as selected by the served user during the service execution triggering.

A basic test scenario is defined as a successful service execution without the service resume via the message channel. In terms of the proposed framework, the scenario implies having the service execution success criteria satisfied during the course of a single interaction performed via the message channel. The basic test scenario is illustrated on Figure 7.3. The execution path is chosen in such a way so it includes all types of the omnichannel-enabled features except the Release feature (Signal error) as the latter is introduced with a single purpose to test the service resume.

The corresponding SIP message sequence is modelled on Figure 7.4. This sequence is used to verify the basic scenario execution by capturing the related traffic.

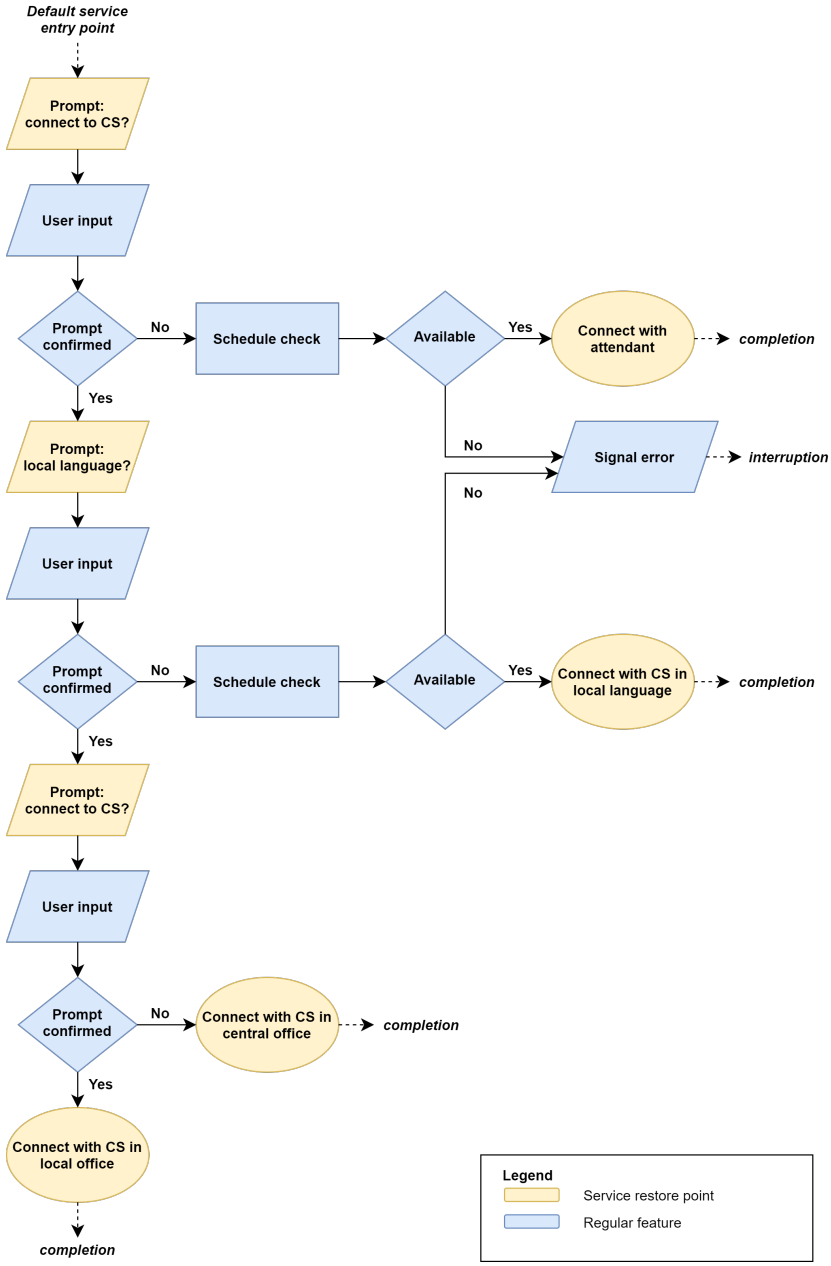


Figure 7.2: Example service logic

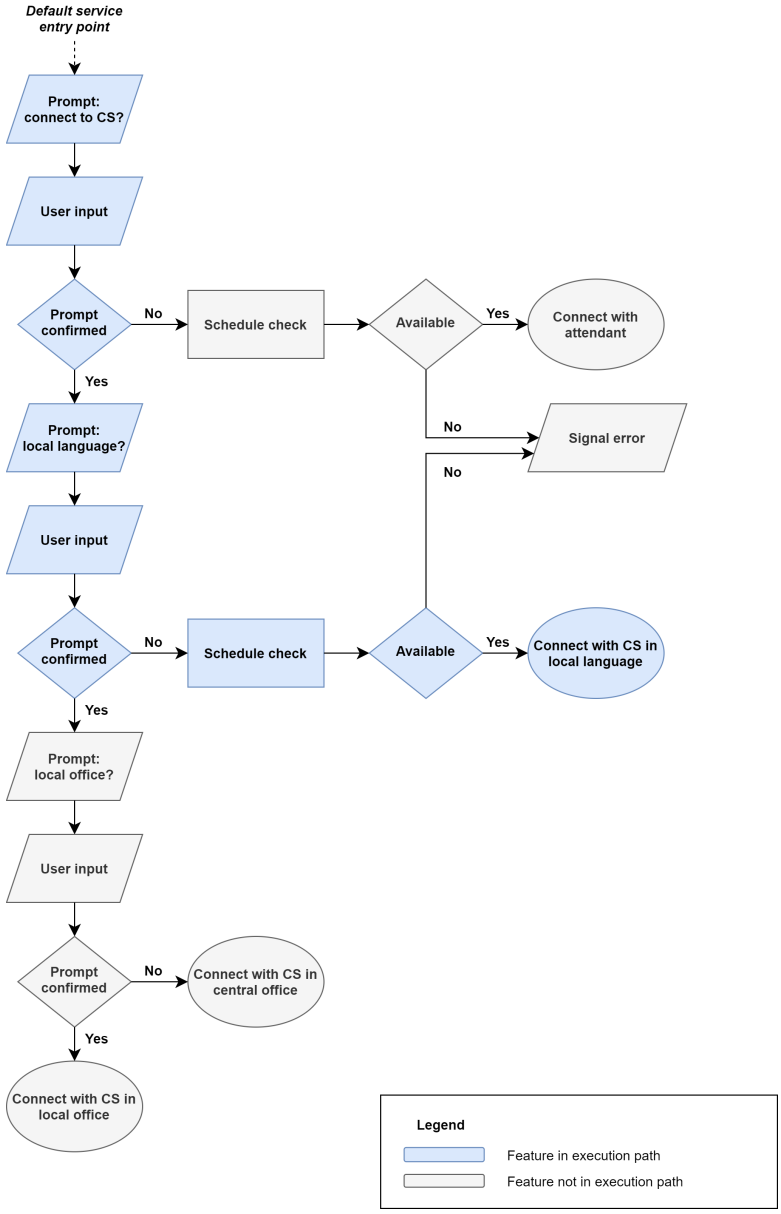


Figure 7.3: Basic test scenario: successful service execution without service resume

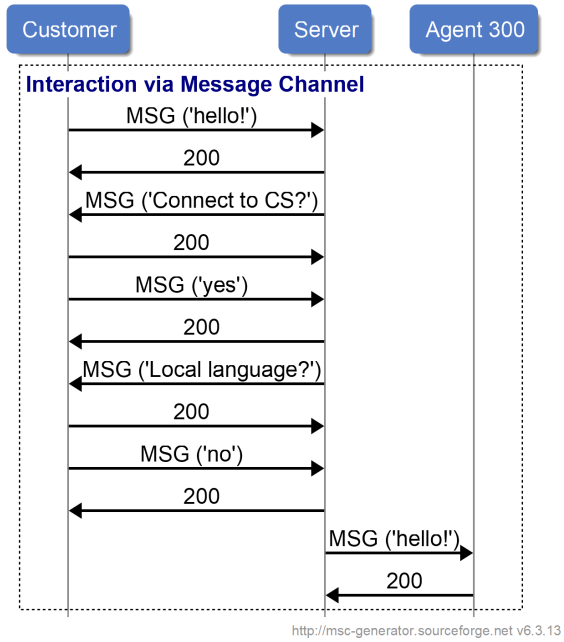


Figure 7.4: Basic test scenario: SIP sequence

7.3 Advanced test scenario

The next iteration of the baseline omnichannel enablement is that of supporting the service resume along with its special case of the channel hopping. An advanced test scenario consists of three interactions within one conversation. First two interactions are conducted via the message channel and are interrupted. The final interaction is conducted via the voice channel and leads to the successful service execution. All the interactions are described separately in the following subsections. SIP message flow for the advanced scenario verification is modelled on Figure 7.8.

7.3.1 The first interaction

The example service is triggered by a chosen user via the message channel. The system indicates that there is no pending conversation for that user-service pair and creates a new one. Then, the service execution starts at the default service entry point. Both timing of the interaction and submitted user input are chosen so that the specific execution path is calculated, see Figure 7.5. The purpose of arranging such an execution path is to test that service resume during the successive interaction works correctly.

The last executed feature during the initial interaction is the Release feature ("Signal error") discussed earlier in the chapter. There are two execution paths leading to that feature, and each of those paths has a different restore point associated with it. It is thus important to check that the service resume resolves such ambiguity correctly and uses the last activated restore point. The dotted arrow on the Figure 7.5 indicates such a restore point which should serve as a new service entry point for the successive interaction.

7.3.2 The second interaction

The service is triggered by the same user via the same channel. The system indicates that there is a pending conversation for that user-service pair with the previous interaction interrupted. Thus, the system creates a new interaction in the scope of the same conversation and performs the service resume. The new service entry point calculated as the last activated restore point during the preceding interaction, see Figure 7.6.

During testing, interactions are initiated one after another with a minimal operational delay. If none of the test parameters is changed, the second interaction is likely to follow the same execution path as the first one. Changing the schedule ccheck parameters or putting even more constraints on the timing choice for the first interaction is impractical. Instead, user input submitted to the first activated prompt is changed. That results in the different execution path selected for the second interaction.

An interruption is introduced during the final step of the flow thus setting this step as a new restore point. The purpose of introducing this interruption is to test the channel hopping in the successive interaction.

7.3.3 The final interaction

The service is triggered by the same user but via the voice channel. The system once again recognizes a pending conversation, creates a new interaction in the scope of that conversation and performs the service resume from the last activated restore point, see Figure 7.7. Once the parties are connected, the example service execution is considered to be completed, and so are the respective interaction and conversation.

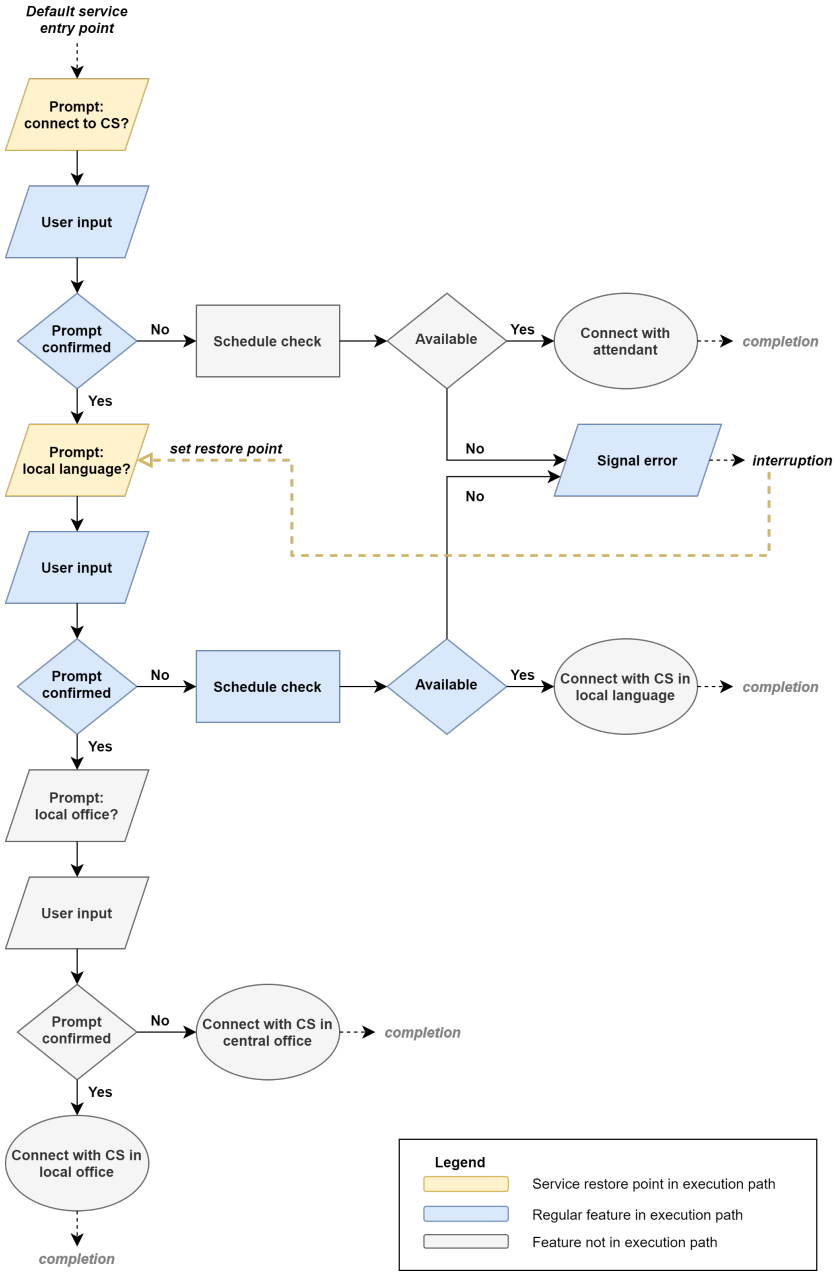


Figure 7.5: Advanced test scenario: initial interaction flow

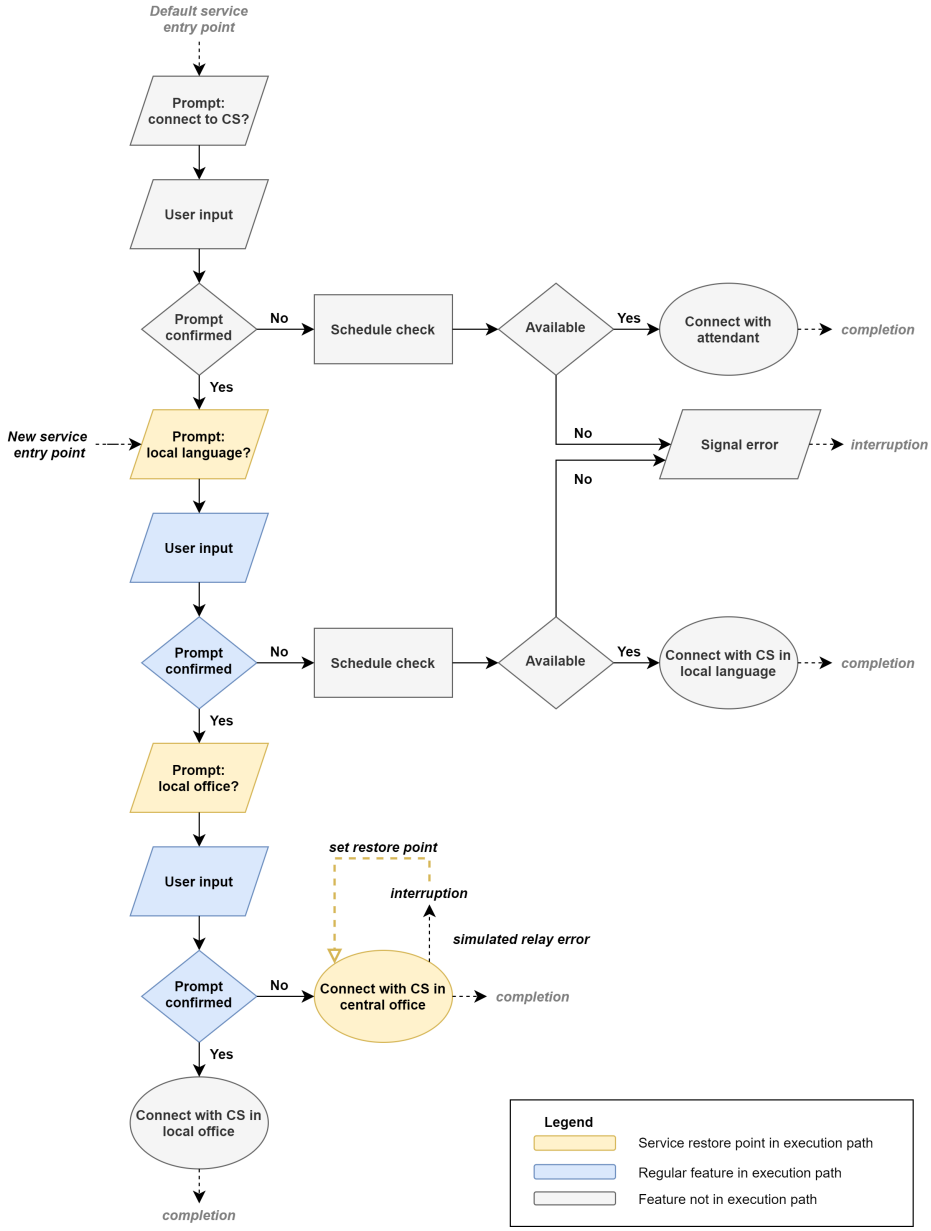


Figure 7.6: Advanced test scenario: second interaction flow

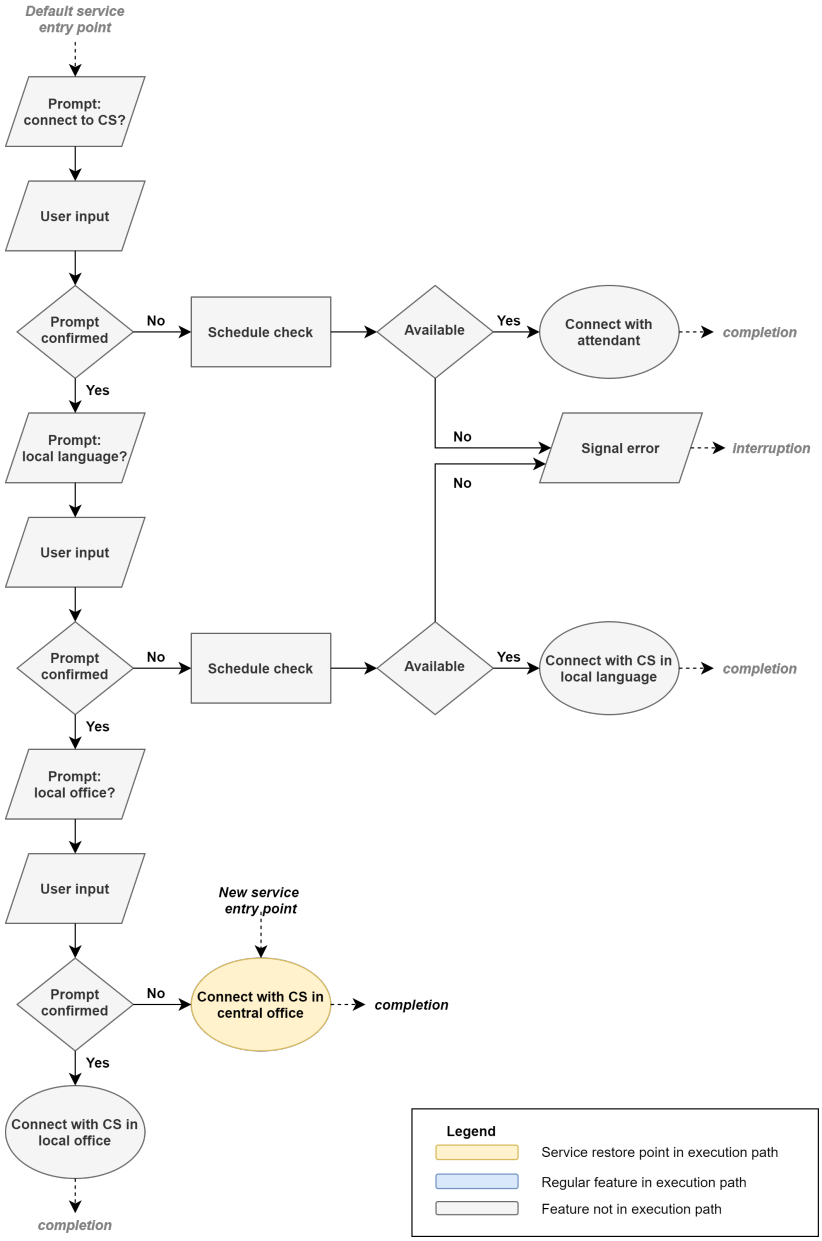


Figure 7.7: Advanced test scenario: final interaction flow

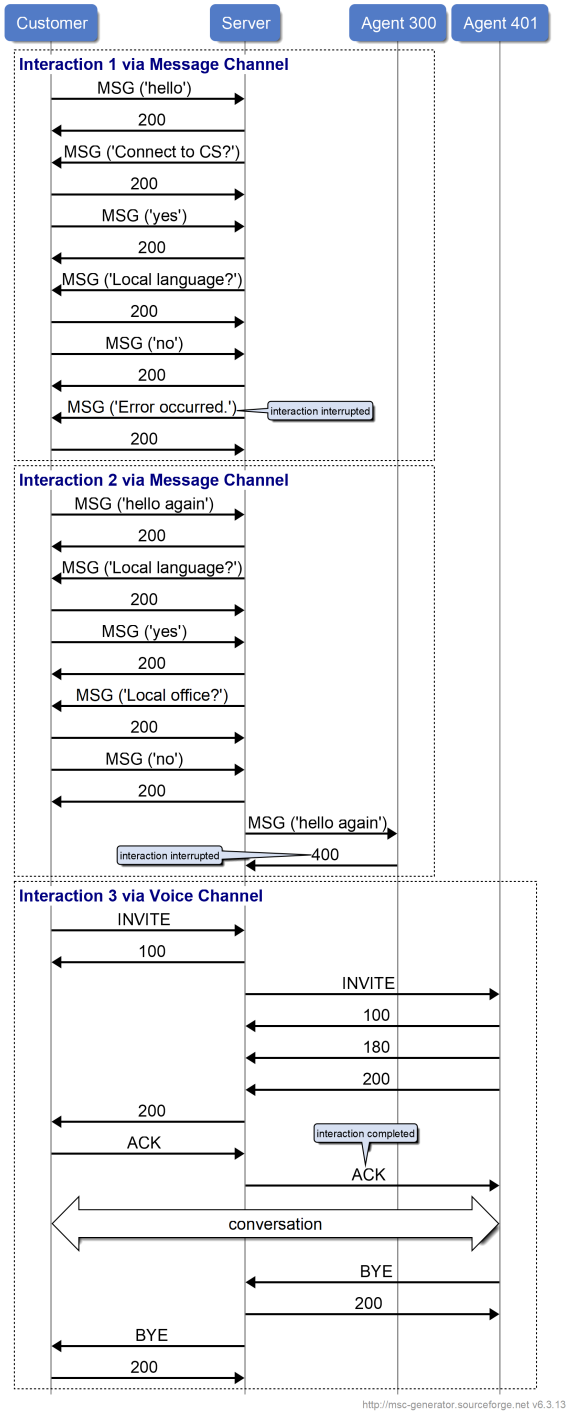


Figure 7.8: Advanced test scenario: SIP sequence

7.4 Test environment

This section provides high-level overview of the test setup used for the prototype validation. Such a setup consists of Gintel framework enhanced with the prototype functionality as well as tools allowing to simulate communication as defined in the test scenarios and verify SIP sequences generated by the framework. The setup is illustrated on Figure 7.9 where introduced enhancements are highlighted with the orange color.

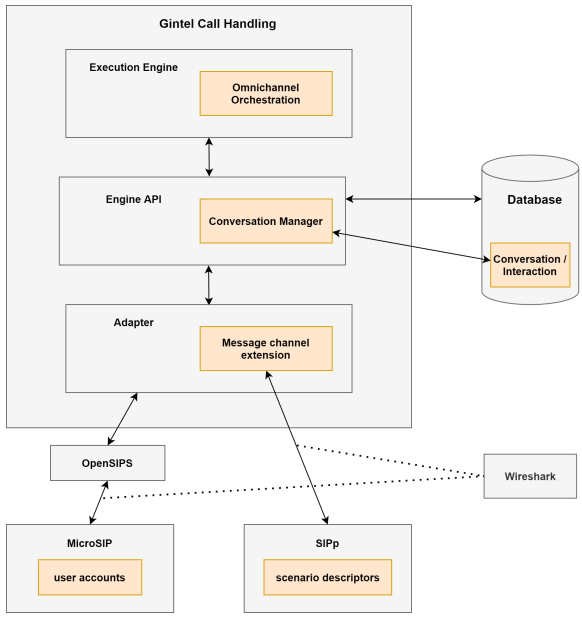


Figure 7.9: Prototype test setup

Gintel framework could be conditionally represented by the following hierarchy of layers:

- *Adapter* is an entry point to the platform performing conversion of received requests into internal events. This layer is extended with support of the message channel.
- *Engine API* performs initial processing of received events and, if necessary, triggers service execution. This layer is extended with Conversation Manager capabilities described in Section 6.1.
- *Execution Engine* performs execution of the features, or components - in Gintel terminology, corresponding to the service triggered. The omnichannel

enablement on this level is done via omnichannel enablement of the dedicated features as well as introducing service resume procedure.

The framework also relies a persistent storage depicted as *Database*. The prototype introduced changes to support storage of conversation-related data which is used by the Conversation Manager. Although in-memory storage is a viable alternative for the prototype, the persistent storage is a practical solution for preserving test data between system restarts as well as for transparently monitoring changes in such data (for example, a change in interaction status or an execution timestamp) and thus assisting in troubleshooting.

As for the tools employed for the test setup, MicroSIP [Mic] and SIPp [SIP] are used to generate communication on behalf of a user/agent and Wireshark [Wir] is used to capture traffic generated by the framework and the tools. OpenSIPS [ope] is playing the role of a SIP registrar/proxy used to enable interconnectivity between the framework and two MicroSIP instances representing a user and an agent when communication takes place via the voice channel. When communication takes place via the message channel, openSIPS is not needed as there is only one instance of SIPP run. The captured traffic is then verified against the modelled SIP Sequences, see Figure 7.4 and Figure 7.8.

MicroSIP is used to generate voice communication corresponding to the final interaction of the advanced test scenario (see Section 7.3). SIPp is used for both the basic test scenario (see Section 7.2 and the two first interactions of the advanced test scenario (see Section 7.3). Custom scenario descriptors are used to both configure generated requests and specify expected ones. Each descriptor can only handle one SIP transaction, whereas the prototype test scenarios consist of multiple transactions. In order to resolve this issue, a custom script was introduced to sequentially run a series of chosen SIPp scenarios. SIPp acts both as a request generator and a universal request sink in the sense it receives requests for both the served user and the chosen agent.

7.5 Implementation

The changes introduced to Gintel framework during the prototype development are illustrated on Figure 7.5 with the changeset element highlighted with orange. The enhancements include both new classes and classes with updated signatures. In the latter case, it is the signature itself which is highlighted whereas the class name stays gray.

The diagram only covers the changes which immediately relate to the omnichannel enablement of the baseline. Those changes can be relatively easily mapped on the

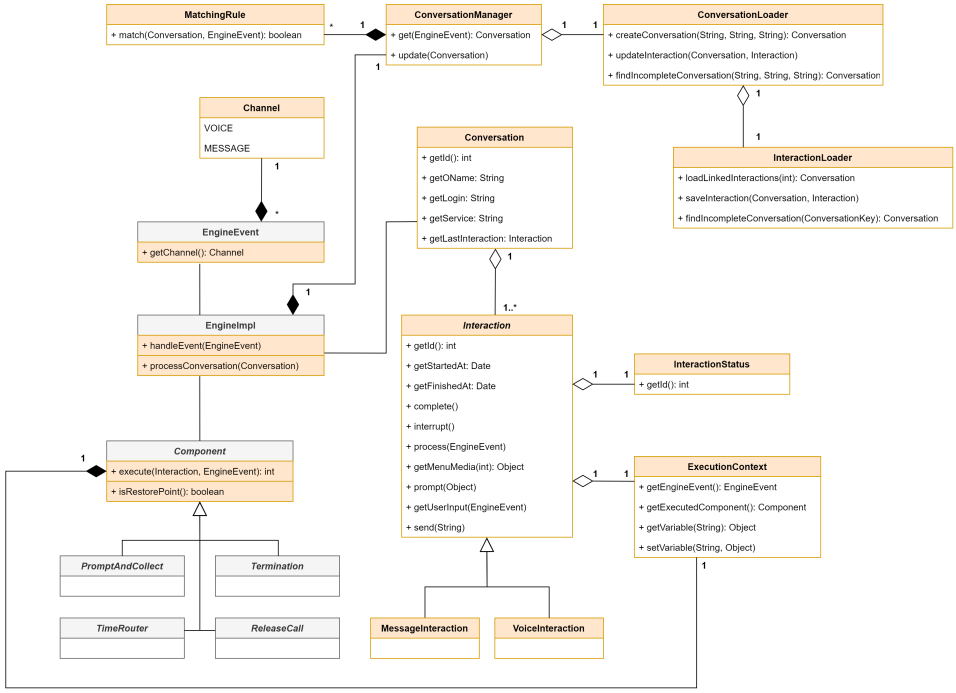


Figure 7.10: Omnichannel enablement class diagram

conceptual framework described in Section 6.1. However, the work conducted during the prototype development extends beyond the presented scope of changes. Due to tight coupling, the baseline required additional code modifications to successfully embed the prototype changes. Although such modifications are not relevant to the topic of omnichannel enablement on their own, they do highlight certain challenges associated with the omnichannel enablement which are going to be discussed in the further section.

The baseline characteristics imposed certain limitations for the framework's omnichannel enablement. With the baseline being a voice-only solution having evolved for almost a decade, the channel-specific logic is scattered across all the layers of the solution and not only components implementing features. Full decoupling implied by the omnichannel enablement proved to be unfeasible in the scope of the prototype development. Re-implementation of the voice channel support from scratch was unfeasible as well due to its complexity. Therefore, voice channel logic has been left intact with *VoiceInteraction* class only dealing with *Interaction* statuses actualization required by the *Conversation Manager*. *MessageInteraction*, on the contrary, was implemented in a new fashion handling channel-specific logic delegated by Components.

Code listings for *Interaction*, *MessageInteraction*, and *ConversationManager* classes are included into Appendices A.1, A.2, and A.3 respectively. In the *Interaction* class, some primitive code such as getters and setters has been omitted.

The very same limitations apply to the classes responsible for service execution: *EngineImpl* and *Component* with its descendants. Thus, new simplified signatures are introduced to handle message channel in an omnichannel manner, whereas voice channel is handled in the legacy manner with minor modifications to support *Interaction* status actualization also for the voice channel. The source codes of these classes cannot be distributed due to legal reasons.

The rest of the classes contain rather auxiliary logic and hence are not distributed either. Data Definition Language (DDL) script for new database structures is covered in Appendix B.

7.6 Testing procedure

The prototype has been tested against both basic and advanced scenarios described in Section 7.2 and Section 7.3 respectively.

For each scenario, a group of SIPp scenario descriptors have been configured to handle distinct transaction types to be executed within the scenario. To accumulate those descriptors in one accumulative scenario, a shell script has been created. Examples of scenario descriptors along with the accumulating execution script are enlisted in Appendix C and cover the advanced test scenario run on the prototype. SIPp configurations for running the basic test scenario are similar.

Before running each scenario, Wireshark was set up to capture traffic on the interface used to connect the prototype and the testing tools. Example of the captured traffic is presented on Figure 7.11 and corresponds to the advanced scenario. The captured traffic is verified against the modelled flow, in the current case against the flow on Figure 7.8. One deviation could be noticed: there is a duplication of messages sent by the framework. An investigation was conducted with a conclusion that such a duplication is a result of a delay occurring on SIPp side while acknowledging the received requests. Thus, the exposed behaviour is actually normal from the perspective of the framework. Otherwise, the traffic captured demonstrated that the prototype correctly handled the service resume procedure in general and the channel hopping in particular.

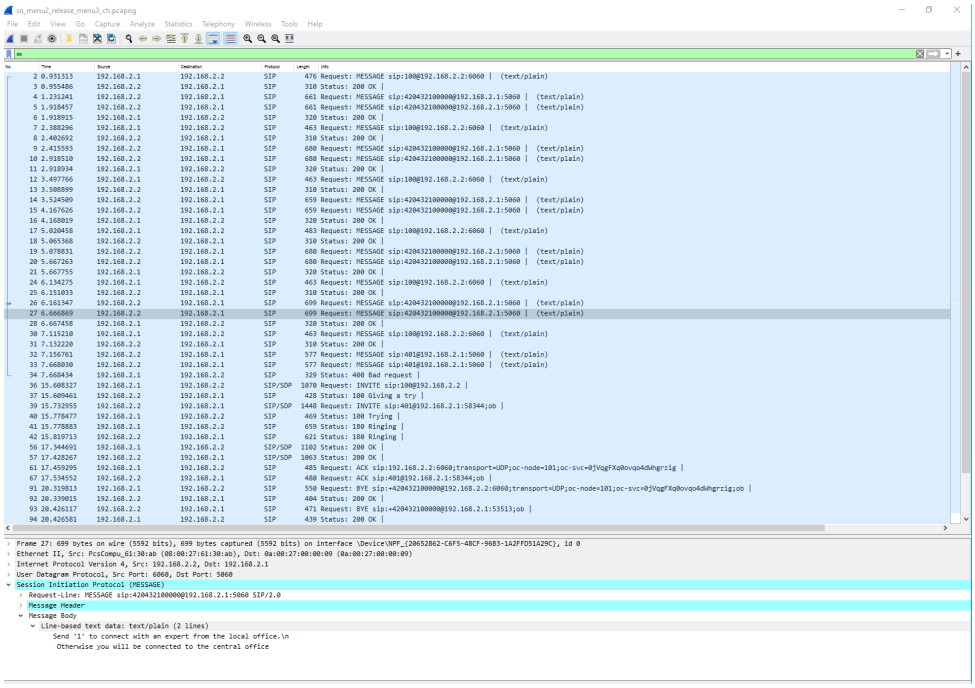


Figure 7.11: Prototype testing as per advanced scenario

Chapter 8
Discussion

The implemented prototype proved feasibility of the framework proposed in Section 6.1. The prototype demonstrated incremental omnichannel enablement approach as applied to the initially voice-only baseline. The baseline's logic has been extended with the support for the message channel for the selected scope of features, with the respective business logic unified across the channels. In addition to that, the common service orchestration has been introduced. The baseline has been also enhanced with the service resume capability as well as its specific case of channel-hopping.

The prototype development, although constrained by its prerequisites such as the baseline, allowed to get additional insights into the topic of the omnichannel enablement and its particular aspects. The insights are not limited to the framework proposed but offer a broader perspective on omnichannel enablement in terms of its associated challenges and prospects.

8.1 Service semantic

One of the cornerstones, as well a non-trivial task, for the omnichannel enablement is to establish channel-agnostic semantic of features and services. For the prototype's sake, certain simplifications were done as the design of a real-life communication service was not an immediate objective of the project. In the example service, the Termination feature was used to establish connection between a user and a specific agent. In terms of the voice channel, "connection established" conceptual event was bound to successful completion of ACK transaction following the initial INVITE transaction (see Figure 7.8) which signifies that the agent picked up and thus was actively involved. In terms of the message channel, the same conceptual event was bound to reception of a successful response code (200) for the initial MESSAGE, which by itself does not imply that the agent has already read the message [IETb]. For the prototyping purposes, the successful relay of the message to a final destination sufficed. However, one could only conditionally equate those

outputs, and an alternative output for the message channel could as well have been chosen as an analogue for the voice channel's output. It is rather a matter of perspective and a conceptual problem to be negotiated by a designer of a real-life communication service.

The problem can be elaborated even further. In the baseline solution, the Termination feature is used not only to connect to a destination but also to control the call flow. Since Gintel is specializing in providing tailored solutions, each feature has an extensive list of parameters through which that feature could be fine-tuned. For example, the Termination feature has an optional parameter regulating maximum call time allowed. Clearly, this parameter cannot be ported to the message channel in a straightforward manner as asynchronous communication is not supposed to be a subject to strict timing conditions by the very definition of such communication. If a feature designer still wants to have such a parameter applied in a cross-channel manner, a new semantic has to be elaborated. However, such a semantic is likely to be less transparent than the one defined for the voice channel.

8.2 Feature-Interaction decoupling

With the number of channel-specific parameters growing, so does the complexity of porting those parameters to new channels. It does not only concern new service semantic which has to be established, but also has a direct impact on the complexity of channel-agnostic interface between Features and Interactions. Design of such an interface has already been discussed as a potential constraint in Section 6.2. The interface is supposed to provide decoupling of business logic implemented by Features and channel-specific logic delegated to Interactions. The interface consists of a set of signatures describing operations and their parameters, with both described in channel-agnostic terms or normalized.

If parameters are not normalized across the channels supported, those parameters could still be included into signatures of the interface. However, that would lead to increased complexity and deteriorated transparency of the signatures, not to mention that the interface itself would not be channel-agnostic anymore. An alternative would be for an Interaction to derive required set of channel-specific parameters in an indirect manner. For example, an Interaction could identify which channel-specific parameters have to be retrieved during a specific operations as defined by the currently executed Feature. Such an approach would compromise Feature-Interaction decoupling and thus would have a negative impact on modularity of the solution. It would also undermine the notion of omnichannel as common processing of communication independent of the channel in use.

8.3 Incremental omnichannel enablement

For the prototype development, the incremental platform-based omnichannel enablement approach was chosen (see Section 2.3). Given the timelines of the project, performing greenfield development was unfeasible whereas the extension of the baseline with omnichannel capabilities allowed to benefit from having the business logic already in place. The initial plan was to perform proper decoupling of that business logic from the channel-specific logic for the selected set of features.

However, with the baseline having evolved as a voice-only solution for over a decade, the omnichannel enablement of the baseline turned out to be far more complicated than initially anticipated as the full stack of the baseline components, and not just features, was a subject to tight-coupling of business and channel-specific logic. Thus, the all-depth decoupling had to be performed. With omnichannel enablement scope extended, such a decoupling might be suboptimal as it is restricted by the backwards compatibility requirements: a new increment should extend, and not limit the existing functionality.

Therefore, forcing omnichannel enablement onto the long-established voice-only solution might not be the best approach to omnichannel enablement. The greenfield approach, however, might serve as a better alternative as it allows to make fundamental architecture changes required for proper omnichannel enablement. Such an approach would also allow to rethink semantic of the services provided and thus handle issues raised in Subsection 8.1 and Subsection 8.2.

8.4 User and service authentication

For both the baseline and the prototype, SIP was a protocol of choice. Hence, the authentication procedure was straightforward and performed uniformly for both the voice and the message channel, via the same SIP headers. However, omnichannel generally implies greater variety in protocols which means that cross-channel authentication should be considered during service design.

Omnichannel is a customer-centric paradigm of communication services (see Section 2.2). Therefore, a problem of correct cross-channel user identification is a cornerstone issue for any omnichannel enablement. Apart from identifying who is triggering a service, it is important to understand which service is requested. Without such authentication, conversation matching is not possible which in its turn disables service resume and channel hopping.

Service authentication procedure is to a great extent determined by a definition of a particular service. In certain cases, an original destination of an incoming communication would serve the purpose. The prototype also relied on that approach

with the "main number" uniquely identifying a service to be provided. However, in more advanced cases, when conceptual matching criteria do not map directly on certain technical parameters, service authentication might potentially employ knowledge management techniques and predictive analysis as discussed in Section 2.1.

8.5 Non-functional aspects

Non-functional aspects of omnichannel enablement have not been in the focus of the project. However, they would play the key role in mapping of the conceptual framework onto a particular architecture for a non-prototype solution.

Special attention should be given to privacy and security aspects which might potentially have a great impact on authentication procedure as well as on general feasibility of advanced omnichannel services.

8.6 Prospects

The ptototype implementation helped to identify some of the advatanges of the framework. Apart from the standard benefits of omnichannel, such as common processing across channels enabled by the decoupling and continuity of service enabled by service resume, the proposed framework allows to envision alternative ways of its exploitation.

The framework offers a set of extension points which could be used by CSP to provide flexible resumable services. The framework explicitly suggests configurability of conversation matching procedure by the means of Matching Rules discussed in more details in Section 6.1. Another potential extension point, although not directly reflected in the proposed framework, is the service resume policy. Initially, a direct approach was implied as it was implemented in the prototype solution: service resume restarts the Service from a Restore Point last activated during the previous Interaction. However, it could also be possible to map all Restore Points to some custom procedures, or service resume policies. Such a policy could trigger execution of a specific execution path which is not a part of main service logic. For example, a special prompt could be used to get a User's confirmation of whether that User prefers to continue with the service resume or restart the Service from scratch. Such added flexibility would allow to design more user-friendly services and thus improve CX

The important benefit of the framework is that of loose coupling between the service resume and channel agnosticism aspects. On practice, that means that an CSP may choose to adopt those aspects separately or even limit adoption to one

aspect only. Although the latter would not result in omnichannel enablement, it could nevertheless optimize the solution in question in terms of either flexibility or maintenance. As for the omnichannel enablement, the framework allows to perform it in incremental manner on the high level with the particular aspects developed as greenfield sub-projects of the overall solution. The loose coupling between aspects allows to minimize adjustments necessary to combine the aspects to the single working solution.

8.7 Summary

The discussion of the omnichannel enablement aspects and prospects leads to a question whether a unified framework for omnichannel communication services is after all viable. The answer to that question is not straightforward. The project has demonstrated certain success in formalization of omnichannel requirements and concepts and even in performing an omnichannel enablement of the existing framework based on the proposed concepts. However, even on the scale of prototype, it has become evident that omnichannel enablement requires re-evaluation of the goals behind it.

Arguably, the primary goal of omnichannel enablement is to offer customers with services perceived as consistent across a variety of channels. Hence, the internal unification is a mere means to achieve the objective. However, trying to force omnichannel enablement onto the solution only for the sake of internal unification is not necessary a wise choice to make. In the prototype's case, the baseline is initially a solution aimed at providing finely tuned specialized services. In its essence, such a definition goes against the notion of omnichannel which implies rather general processing across channels supported. Such a mismatch has caused challenges during establishing common feature semantic and decoupling. Persisting with such enablement would not result in a viable solution as it inevitably leads to compromising either principles of omnichannel, or goals of the baseline.

There is no evidence, however, that the framework cannot be considered viable. On the contrary, the prototype has proved that the proposed concepts and approaches could be to a great extent successfully applied, even constrained by the baseline and scope choices. If services to be supported are defined as truly omnichannel in terms of their semantic, it is anticipated to automatically resolve the majority of issues associated with internal logic unification. That, however, is likely to come at a cost of generalization of a solution.

Chapter 9
Conclusion

This thesis represents a fair attempt to approach the topic of omnichannel enablement as applied to the domain of communication services. Despite the market appeal of omnichannel, the knowledge context behind the topic lacks consolidation as the respective academic research is scarce and technical specifications of commercial offerings remain to a great extent proprietary and not available to the general public.

In scope of this project, the study of disparate research and market offerings has been conducted which allowed to build understanding of the implications of omnichannel enablement for a communication services platform. Based on this understanding, the requirements for unified framework have been formulated followed by the conceptual model of such a framework. The framework has been validated by the means of prototyping which both has confirmed the framework's feasibility and allowed to take a closer look at different aspects associated with omnichannel enablement.

The results of the project should be treated as the first approximation to consolidation and formalization of omnichannel implications when building a unified platform for communication services. Future research should be directed at particular aspects as highlighted during the discussion stage. As complex as it is, omnichannel enablement is the prerequisite of a successful digital transformation, and as such it is going to remain on the wish list of CSPs in the foreseeable future.

References

[BCG+05] G. W. Bond, E. Cheung, H. H. Goguen, K. J. Hanson, D. Henderson, G. M. Karam, K. H. Purdy, T. M. Smith, and P. Zave. Experience with Component-Based Development of a Telecommunication Service. In G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. Szyperski, and K. Wallnau, editors, *Component-Based Software Engineering*, pages 298–305, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[BCP+04] G. Bond, E. Cheung, K. Purdy, P. Zave, and J. Ramming. An Open Architecture for Next-Generation Telecommunication Services. *ACM transactions on Internet technology*, 4(1):83–123, 2004.

[BGK16] A. Bolton, L. Goosen, and E. Kritzinger. Enterprise Digitization Enablement Through Unified Communication amp; Collaboration. In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, SAICSIT '16, New York, NY, USA, 2016. Association for Computing Machinery.

[BKv+20] B. Burke, F. Karamouzis, G. van der Heiden, A. Chandrasekaran, D. Cearley, B. Willemsen, B. Stewart, R. Krikken, J. Heiser, E. Brethenoux, J. Wong, M. Chiu, M. Duerst, D. Scheibenreif, M. Bhat, T. Harvey, N. Sturgill, L. Shotton, S. Stoudt-Hansen, G. Alvarez, D. Gaughan, A. White, D. Smith, E. Anderson, D. Wright, and Y. Natis. Top strategic technology trends for 2021. https://www.gartner.com/document/3991906, October 2020.

[Boc17] L. Bocklund. The multichannel contact center becomes "omnichannel". *Contact Center Pipeline*, September 2017.

[CC17] P. Clark-Dickson and J. Cox. Orchestrating the Omnichannel Customer Experience. Whitepaper, Ovum, April 2017.

[CGL+93] E.J. Cameron, N. Griffeth, Y.-J. Lin, M.E. Nilson, W.K. Schnure, and H. Velthuijsen. A feature-interaction benchmark for IN and beyond. 31(3):64–69, 1993.

[Cisa] Cisco. Omnichannel Contact Center. https://cjp.broadsoft.com/contact-center-performance-solutions/omni-channel-contact-center/l.

[Cisb] Cisco. Unified Communications and Collaboration. https://www.cisco.com/c/en/us/products/unified-communications/index.html.

[Cox18] J. Cox. 2019 Trends to Watch: Customer Engagement Platforms. Whitepaper, Ovum, November 2018.

[CS09] E. Cheung and T.M Smith. Experience with modularity in an advanced teleconferencing service deployment. In *2009 31st International Conference on Software Engineering - Companion Volume*, pages 39–49. IEEE, 2009.

[Fin20] E. Finegold,. Future customer experience: From digital to omnichannel. Research report, TM Forum, March 2020.

[FLB+17] S. Filiposka, R. Lapacz, M. Balcerkiewicz, F. Wein, and J. Sobieski. Transforming silos to next-generation services. In *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*, pages 745–750, 2017.

[FMWS16] S. Filiposka, A. Mishev, F. Wein, and J. Sobieski. Customer-Centric Service Provider Architecture for the R E Community. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 596–601, 2016.

[GCM12] N. Ghiata, V. Cretu, and M. Marcu. Context-aware unified communication. In *2012 7th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 297–301, 2012.

[Gena] Genesys. Context Services Developer's Guide 8.5.2. https://docs.genesys.com/Documentation/CS/8.5.2/Developer/Welcome. Accessed: 26.03.2021.

[Genb] Genesys. Conversation Manager Overview 8.5. https://docs.genesys.com/Documentation/CM/8.5/Overview/Welcome. Accessed: 26.03.2021.

[Genc] Genesys. Conversation Rules Templates Guide 8.5.1. https://docs.genesys.com/Documentation/GRS/8.5.1/CR/Welcome. Accessed: 26.03.2021.

[Gend] Genesys. Genesys Engage | Customer Experience Platform. https://www.genesys.com/genesys-engage. Accessed: 26.03.2021.

[HB10] D. Hong and A. Brinsmead. Optimizing Customer Service in a Multi-Channel World. Whitepaper, Ovum, October 2010.

[HR04] H. Lei and A. Ranganathan. Context-aware unified communication. In *IEEE International Conference on Mobile Data Management, 2004. Proceedings. 2004*, pages 176–186, 2004.

[HSD+02] H.Lei, D. M. Sow, J. S. Davis, G. Banavar, and M. R. Ebling. The Design and Applications of a Context Service. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):45–55, 2002.

[IETa] IETF. RFC 3261 - SIP: Session Initiation Protocol. https://tools.ietf.org/html/rfc3261. accessed: 25.03.2021.

[IETb] IETF. RFC 3428 - Session Initiation Protocol (SIP) Extension for Instant Messaging. https://datatracker.ietf.org/doc/html/rfc3428. accessed: 1.04.2021.

[JZ98] M. Jackson and P. Zave. Distributed feature composition: a virtual architecture for telecommunications services. *IEEE transactions on software engineering*, 24(10):831–847, 1998.

[Lar20] Vladislava Larina. Unified framework for omnichannel communication services. Project report in TTM4502, Department of Information Security and Communication Technology, NTNU – Norwegian University of Science and Technology, December 2020.

[McE20] T. McElligott. Committing to omnichannel. https://inform.tmforum.org/insights/2020/10/committing-to-omnichannel//, October 2020. accessed: 21.01.2021.

[Mic] MicroSIP. Open source portable SIP softphone for Windows based on PJSIP stack. https://www.microsip.org/.

[Mit] Mitel. Omnichannel Enterprise Call Center and Contact Center. https://www.mitel.com/products/applications/contact-center/other.

[ML20] B. Manusama and N. LeBlanc. Market guide for digital customer service and support technologies. https://www.gartner.com/document/3982236, March 2020.

[MRE+21] B. Manusama, M. Revang, B. Elliot, S. Blood, N. LeBlanc, P. Rathnayake, D. Alvord, J. Robinson, E. Potosky, S. Dibble, J. Davies, and Customer Service and Support Research Team. 2021 strategic roadmap for customer service and support: 10 dilemmas. https://www.gartner.com/document/3996004, January 2021.

[NCT+20] A. Nandan, M. Cana, K. Takiishi, P. Liu, and T. Chamberlin. Predicts 2021: CSP Technology and Operations Strategy. https://www.gartner.com/document/3993798, December 2020.

[NGO+19] Y. Natis, D. Gaughan, M. O'Neill, B. Lheureux, and M. Pezzini. Innovation insight for packaged business capabilities and their role in the future composable enterprise. https://www.gartner.com/document/3976170, December 2019.

[ope] openSIPS. OpenSIPS About. https://opensips.org/About/About.

[OU20] D. O'Connel and L. Unden-Farboud. Market guide for communications platform as a service. https://www.gartner.com/document/3991743, October 2020.

[PMF+17] J. Pérez, M. Murray, J. Fluker, D. Fluker, and Z. Bailes. Connectivity and Continuity: New Fronts in the Platform War. *Communications of the Association for Information Systems*, 40, April 2017.

[PMP18] D. Peras, R. Mekovec, and R. Picek. Influence of gdpr on social networks used by omnichannel contact center. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1132–1137, 2018.

[PPM18] R. Picek, D. Peras, and R. Mekovec. Opportunities and challenges of applying omnichannel approach to contact center. In *2018 4th International Conference on Information Management (ICIM)*, pages 231–235, 2018.

[RT09] K. Riemer and S. Taing. Unified Communications. *Business & information systems engineering*, 1(4):326–330, 2009.

[SIP] SIPp. SIPp v3.3 documentation. http://sipp.sourceforge.net/doc3.3/reference.html.

[Smi10] T. Smith. Reusable features for VoIP service realization. In *Principles, Systems and Applications of IP Telecommunications*, IPTComm '10, pages 42–47. ACM, 2010.

[TFO+20] C. Trueman, M. Fernandez, D. O'Connell, R. Benitez, and P. Sheth. Critical capabilities for unified communications as a service, worldwide. https://www.gartner.com/document/3993202, November 2020.

[TLCZ14] A. D. Tesfamicael, V. Liu, W. Caelli, and J. Zureo. Implementation and Evaluation of Open Source Unified Communications for SMBs. In *2014 International Conference on Computational Intelligence and Communication Networks*, pages 1243–1248, 2014.

[TMF18] GB994 Omni Channel Guidebook. Best Practice R18.0.1, TM Forum, August 2018.

[Tur15] A. Turner. Omnichannel: navigating new territory. Research report, TM Forum, November 2015.

[Tur17] A. Turner. Customer centricity: Creating the digital experience. Research report, TM Forum, September 2017.

[Wie14] R. J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering*. Berlin, Heidelberg, 2014 edition, 2014.

[Wir] Wireshark. About Wireshark. https://www.wireshark.org/index.html#aboutWS.

[WSP20] J. Wong, D. Scheibenreif, and G. Phifer. Transcend omnichannel thinking and embrace multiexperience for improved cx. https://www.gartner.com/document/3981020, February 2020.

[Zav09] P. Zave. Modularity in Distributed Feature Composition. *Software Requirements and Design: The Work of Michael Jackson*, February 2009.

[Zav11] P. Zave. Mid-call, multi-party, and multi-device telecommunication features and their interactions. In *Proceedings of the 5th International Conference on principles, systems and applications of IP telecommunications*, IPTcomm '11. ACM, 2011.

[ZC09] P. Zave and E. Cheung. Compositional Control of IP Media. *IEEE Transactions on Software Engineering*, 35(1):46–66, 2009.

[ZJ02] P. Zave and M. Jackson. A Call Abstraction for Component Coordination. 66(4):36–55, 2002.

[ZJ10] N. P. Zhang and J. S. Jiang. A Reference Model of Unified Communication Based on SIP/SIMPLE. *Applied mechanics and materials*, 20-23:232–235, 2010.

Appendix

Appendix A

A.1 Conversation Manager

```
1  package com.gintel.omni.conversation;
2
3  import com.gintel.common.engine.constants.EventType;
4  import com.gintel.common.engine.interfaces.EngineEvent;
5  import com.gintel.common.logger.LoggerFactory;
6  import com.gintel.common.logger.interfaces.Logger;
7  import com.gintel.omni.interaction.Channel;
8  import com.gintel.omni.interaction.Interaction;
9  import com.gintel.omni.interaction.InteractionStatus;
10 import org.springframework.util.CollectionUtils;
11
12 import javax.sql.DataSource;
13 import java.util.*;
14
15 public class ConversationManager {
16     private static Logger logger = LoggerFactory.getLogger(
           ConversationManager.class);
17
18     public static final String KEY_DELIMETER = ":";
19
20     private Map<String, Conversation> localStore = new
           HashMap<>();
21
22     private List<MatchingRule> matchingRules = new ArrayList
           <>();
23
24     private ConversationLoader dbLoader;
25
```

```
26      private String calculateKey(String userId, String orgId,
            String serviceId) {
27          return new StringBuilder()
28                  .append(userId)
29                  .append(KEY_DELIMETER)
30                  .append(orgId)
31                  .append(KEY_DELIMETER)
32                  .append(serviceId).toString();
33      }
34
35      public ConversationManager(DataSource ds, List<
            MatchingRule> matchingRules) {
36          dbLoader = new ConversationLoader(ds);
37          if (!CollectionUtils.isEmpty(matchingRules)) {
38              this.matchingRules = matchingRules;
39          }
40          logger.info("Initialized with ds=%1s, matching rules
                =%2s", ds, matchingRules);
41      }
42
43      public void update(Conversation conversation) {
44          String orgName = conversation.getOname();
45          String userName = conversation.getLogin();
46          String serviceName = conversation.getService();
47
48          String key = calculateKey(orgName, userName,
                serviceName);
49          localStore.put(key, conversation);
50          dbLoader.updateInteraction(conversation,
                conversation.getLastInteraction());
51          logger.info("Conversation %1s updated", conversation
                .getId());
52      }
53
54      public Conversation get(EngineEvent event) {
55          Conversation result = null;
56
57          String userId = event.getUserId();
58          String orgId = event.getOrganizationId();
59          String serviceId = event.getServiceId();
60          String key = calculateKey(userId, orgId, serviceId);
```

```
61          Conversation conversation = localStore.get(key);
62          logger.info("Found in local store " + conversation);
63          if (conversation == null) {
64              conversation = dbLoader.
                    findIncompleteConversation(orgId, userId,
                    serviceId);
65              logger.info("Found in db " + conversation);
66          }
67
68          if (conversation != null) {
69              Interaction lastInteraction = conversation.
                    getLastInteraction();
70              InteractionStatus lastInteractionStatus =
                    lastInteraction.getStatus();
71              logger.info("Last interaction has status " +
                    lastInteractionStatus);
72              switch (lastInteractionStatus){
73                  case COMPLETED: break;
74                  case IN_PROGRESS:
75                      Channel ch = lastInteraction.getChannel
                            ();
76                      if (!ch.equals(event.getChannel())) {
77                          return null;
78                      }
79                      result = conversation;
80                      break;
81                  default:
82                      boolean isMatch = false;
83                      for (MatchingRule rule : matchingRules)
                            {
84                          isMatch = rule.match(conversation,
                                event);
85                          if (!isMatch) {
86                              logger.info("Failed matching by
                                    rule " + rule);
87                              break;
88                          }
89                      }
90                      result = isMatch? conversation : null;
91                      if (result != null) {
```

```
92                          if (EventType.MESSAGE_ACK.equals(
                                event.type)) {
93                              logger.info("Message ACK doesn't
                                    trigger a new interaction
                                   creation");
94                              break;
95                          }
96                          Interaction newInteraction =
                                Interaction.
                                createResumeInteraction(event.
                                getChannel(), lastInteraction);
97                          result.setLastInteraction(
                                newInteraction);
98                      }
99                      break;
100             }
101         }
102         if (result == null && event.isInitial()) {
103             result = dbLoader.createConversation(orgId,
                    userId, serviceId);
104             result.setLastInteraction(Interaction.
                    createInteraction(event.getChannel()));
105             localStore.put(key, result);
106             logger.info("Created new conversation " +
                    conversation);
107         }
108         return result;
109     }
110
111 }
```

A.2 Interaction

```
1  package com.gintel.omni.interaction;
2
3  import com.gintel.common.engine.constants.EventType;
4  import com.gintel.common.engine.constants.ReleaseCode;
5  import com.gintel.common.engine.interfaces.EngineEvent;
6  import com.gintel.common.logger.LoggerFactory;
7  import com.gintel.common.logger.interfaces.Logger;
8  import com.gintel.common.service.component.Component;
9  import com.gintel.common.service.component.Service;
10
11 import java.util.Date;
12
13 public abstract class Interaction {
14
15 /*
16 Omitted code.
17 */
18     public Interaction() {
19         startedAt = new Date(System.currentTimeMillis());
20         status = InteractionStatus.IN_PROGRESS;
21         executionContext = new ExecutionContext();
22     }
23
24     public abstract Channel getChannel();
25
26     public static Interaction createInteraction(Channel
           channel) {
27         switch (channel) {
28             case MESSAGE:
29                 return new MessageInteraction();
30             case VOICE:
31                 return new VoiceInteraction();
32             default:
33
34                 return null;
35         }
36     }
37
38     public static Interaction createResumeInteraction(
           Channel channel, Interaction interruptedInteraction)
```

```
        {
39          Interaction interaction = createInteraction(channel)
                ;
40          int resumePoint = interruptedInteraction.getCid();
41          interaction.setCid(resumePoint);
42          return interaction;
43      }
44
45      public void finalize(InteractionStatus status) {
46          this.finishedAt = new Date(System.currentTimeMillis
                ());
47          this.status = status;
48      }
49
50      public void setCurrentComponent(Component component) {
51
52          this.executionContext.setExecutedComponent(component
                );
53      }
54
55      public Component getCurrentComponent() {
56          Component currentComponent = this.executionContext.
                getExecutedComponent();
57          logger.info("getCurrentComponent: retrieved from
                exec context = " + currentComponent);
58          if (currentComponent == null) {
59              Service service = getService();
60              if (service != null) {
61                  int cid = this.cid != DUMMY_ID ? this.cid :
                        service.getInitialComponent();
62                  logger.info("getCurrentComponent:
                        restorePoint = " + this.cid);
63                  logger.info("getCurrentComponent: calculated
                         cid = " + cid);
64                  currentComponent = service.getComponent(cid)
                        ;
65              }
66          }
67          return currentComponent;
68      }
69
```

```
70      public Service getService() {
71          return executionContext.getService();
72      }
73
74      public void interrupt () {
75          interrupt(null);
76      }
77
78      public void interrupt(ReleaseCode releaseCode) {
79          if (status.equals(InteractionStatus.COMPLETED)) {
80              logger.warning("Ignored attempt to interrupt
                    completed interaction");
81              return;
82          }
83          status = InteractionStatus.INTERRUPTED;
84          finishedAt =  new Date(System.currentTimeMillis());
85      };
86
87      public abstract void send(String to);
88
89      public void complete() {
90          if (status.equals(InteractionStatus.INTERRUPTED)) {
91              logger.warning("Ignored attempt to complete
                    interrupted interaction");
92              return;
93          }
94          status = InteractionStatus.COMPLETED;
95          finishedAt =  new Date(System.currentTimeMillis());
96      };
97
98      public abstract Object getMenuMedia(int mediaId);
99
100     public abstract void prompt(Object menuMedia);
101
102     public abstract EventType interpretEvent(EventType
            triggeringEvent);
103
104     public abstract String getUserInput(EngineEvent event);
105
106     public abstract void resetExecState();
107
```

```
108     public abstract void process(EngineEvent event);
109 }
```

A.3 MessageInteraction

```
1  package com.gintel.omni.interaction;
2
3  import com.gintel.common.adapter.cc.FunctionMessage;
4  import com.gintel.common.adapter.interfaces.NetworkAdapter;
5  import com.gintel.common.engine.cc.EventMessage;
6  import com.gintel.common.engine.constants.EventType;
7  import com.gintel.common.engine.constants.ReleaseCode;
8  import com.gintel.common.engine.interfaces.EngineEvent;
9  import com.gintel.common.logger.LoggerFactory;
10 import com.gintel.common.logger.interfaces.Logger;
11 import com.gintel.common.service.component.Service;
12 import com.gintel.omni.stubs.ErrorMessageRepository;
13 import com.gintel.omni.stubs.MessageMediaRepository;
14
15 import java.util.HashMap;
16 import java.util.Map;
17
18 public class MessageInteraction extends Interaction {
19
20     private static Logger logger = LoggerFactory.getLogger(
          MessageInteraction.class);
21
22     public enum State {
23         INIT(new HashMap<EventType, EventType>() { {
24                 put(EventType.MESSAGE, EventType.
                      INCOMING_CALL);
25             }
26         }),
27         PROMPT_SENT(new HashMap<EventType, EventType>() {
28             {
29                 put(EventType.MESSAGE_ACK, EventType.ANSWER)
                      ;
30                 put(EventType.MESSAGE, EventType.
                      PROMPT_AND_COLLECT_ENDED);
31
32             }
33         }),
34         INTERRUPTED(new HashMap<EventType, EventType>() {
35             {
```

```
36                  put(EventType.MESSAGE_ACK, EventType.ANSWER)
                        ;
37              }
38          });
39
40          private Map<EventType, EventType> eventMapping;
41
42          State(Map<EventType, EventType> eventMapping) {
43              this.eventMapping = eventMapping;
44          }
45
46          public EventType normalizeEvent(EventType
                triggerType) {
47              return eventMapping.get(triggerType);
48          }
49      }
50
51      public MessageInteraction() {
52          ExecutionContext executionContext =
                getExecutionContext();
53          executionContext.setVariable("state", State.INIT);
54      }
55
56      @Override
57      public ExecutionContext getExecutionContext() {
58          return super.getExecutionContext();
59      }
60
61      @Override
62      public Channel getChannel() {
63          return Channel.MESSAGE;
64      }
65
66      @Override
67      public Service getService() {
68          return executionContext.getService();
69      }
70
71      @Override
72      public void send(String to) {
73          EngineEvent event = executionContext.getEvent();
```

```
74          if (EventType.MESSAGE.equals(event.type)) {
75              EventMessage em  = (EventMessage) event;
76              NetworkAdapter networkAdapter = executionContext
                    .getNetworkAdapter();
77              String content = (String) executionContext.
                    getVariable("userInputForRetran");
78              logger.info("send: interaction id=" + getId());
79              networkAdapter.handleFunction(new
                    FunctionMessage(em.getAppSessionId(), em.
                    getOrigRequest(), to, content));
80          }
81      }
82
83      @Override
84      public Object getMenuMedia(int mediaId) {
85          return MessageMediaRepository.getMessageMedia(
                mediaId);
86      }
87
88      @Override
89      public void prompt(Object menuMedia) {
90          EngineEvent event = executionContext.getEvent();
91          if (EventType.MESSAGE.equals(event.type)) {
92              EventMessage em  = (EventMessage) event;
93              NetworkAdapter networkAdapter = executionContext
                    .getNetworkAdapter();
94              logger.info("prompt: interaction id=" + getId())
                    ;
95              executionContext.setVariable("state", State.
                    PROMPT_SENT);
96              networkAdapter.handleFunction(new
                    FunctionMessage(em.getAppSessionId(), em.
                    getOrigRequest(), null, (String) menuMedia));
97          }
98      }
99
100     @Override
101     public EventType interpretEvent(EventType triggerType) {
102         State state = (State) executionContext.getVariable("
                state");
103         logger.info("interpretEvent: state = " + state);
```

```
104             return state.normalizeEvent(triggerType);
105         }
106 
107         @Override
108         public String getUserInput(EngineEvent event) {
109             if (EventType.MESSAGE.equals(event.type)) {
110                 logger.info("getUserInput from event " +  event)
                        ;
111                 return ((EventMessage)event).getContent();
112             }
113             return "";
114         }
115 
116         @Override
117         public void resetExecState() {
118             executionContext.setVariable("state", State.INIT);
119         }
120 
121         @Override
122         public void process(EngineEvent event) {
123             State state = (State) executionContext.getVariable("
                    state");
124             if (State.INIT.equals(state) && EventType.MESSAGE.
                    equals(event.type)) {
125                 String content = ((EventMessage) event).
                        getContent();
126                 logger.info("processInput: saving user input for
                         retransmission:" + content);
127                 executionContext.setVariable("userInputForRetran
                        ", content);
128             }
129         }
130 
131         @Override
132         public void interrupt(ReleaseCode code) {
133             super.interrupt(code);
134             if (code == null) {
135                 return;
136             }
137             String errorMessage = ErrorMessageRepository.
                    getErrorMessage(code.intValue());
```

```
138             EngineEvent event = executionContext.getEvent();
139             if (EventType.MESSAGE.equals(event.type)) {
140                 EventMessage em  = (EventMessage) event;
141                 NetworkAdapter networkAdapter = executionContext
                        .getNetworkAdapter();
142                 logger.info("interrupt: interaction id = " +
                        getId());
143                 networkAdapter.handleFunction(new
                        FunctionMessage(em.getAppSessionId(), em.
                        getOrigRequest(), null, (String) errorMessage
                        ));
144             }
145         }
146 }
```

Appendix

Appendix B

B.1 Conversation DML (MySQL)

```
1  CREATE TABLE `Conversation` (
2      `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '
           Conversation id',
3      `oname` VARCHAR(64) NOT NULL COMMENT 'Organization name'
           COLLATE 'utf8mb4_unicode_ci',
4      `login` VARCHAR(20) NOT NULL COMMENT 'Login' COLLATE '
           utf8mb4_unicode_ci',
5      `service` VARCHAR(64) NOT NULL COMMENT 'Service' COLLATE
           'utf8mb4_unicode_ci',
6      PRIMARY KEY (`id`) USING BTREE,
7      INDEX `END_USER_FK` (`oname`, `login`) USING BTREE,
8      INDEX `SERVICE_FK` (`service`) USING BTREE,
9      CONSTRAINT `END_USER_FK` FOREIGN KEY (`oname`, `login`)
           REFERENCES `easydb`.`EndUser` (`oname`, `login`) ON
           UPDATE CASCADE ON DELETE RESTRICT,
10     CONSTRAINT `SERVICE_FK` FOREIGN KEY (`service`)
           REFERENCES `easydb`.`Service` (`sno`) ON UPDATE
           CASCADE ON DELETE RESTRICT
11 )
12 COMMENT='Omnichannel support'
13 COLLATE='utf8mb4_unicode_ci'
14 ENGINE=InnoDB
15 AUTO_INCREMENT=220
16 ;
```

B.2 Interaction DML (MySQL)

```
CREATE TABLE `Interaction` (
    `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT 'Interaction id',
    `conversation_id` INT(10) UNSIGNED NOT NULL COMMENT 'Conversation id',
    `channel` SMALLINT(6) NOT NULL COMMENT 'Channel type',
    `startedAt` TIMESTAMP NULL DEFAULT NULL COMMENT 'Start timestamp',
    `finishedAt` TIMESTAMP NULL DEFAULT NULL COMMENT 'End timestamp',
    `status` SMALLINT(6) NOT NULL COMMENT 'Status',
    `cid` INT(64) NULL DEFAULT NULL COMMENT 'Executing component id',
    PRIMARY KEY (`id`) USING BTREE,
    INDEX `CONVERSATION_ID_FK` (`conversation_id`) USING BTREE,
    CONSTRAINT `CONVERSATION_ID_FK` FOREIGN KEY (`conversation_id`) REFERENCES `easydb`.`Conversation` (`id`) ON UPDATE CASCADE ON DELETE RESTRICT
)
COMMENT='Omnichannel support'
COLLATE='utf8mb4_unicode_ci'
ENGINE=InnoDB
AUTO_INCREMENT=216
;
```

Appendix

Appendix C

C.1 sipp/advanced_scenario/send_init.xml

```
<scenario name="Send initial request for service">
  <send>
    <![CDATA[
    MESSAGE sip:100@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[
        branch]
    Max-Forwards: 70
    From: 420432100000 <sip:420432100000@[local_ip]:[
        local_port]>;tag=[pid]SIPpTag00[call_number]
    To: 100 <sip:100@[remote_ip]:[remote_port]>
    Call-ID: [call_id]
    CSeq: 1 MESSAGE
    Content-Type: text/plain
    Content-Length: [len]
    X-OMNI-User: 420432100000
    X-OMNI-Organization: VladaCompany
    X-OMNI-Service: VladaOutbound

    Hello!
    ]]>
  </send>

  <recv response="200">
  </recv>

</scenario>
```

C.2 sipp/advanced_scenario/send_ack.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "sipp.dtd">
<scenario name="Acknowledge request">

  <recv request="MESSAGE">
  </recv>

  <send>
    <![CDATA[

      SIP/2.0 200 OK
      [last_Via:]
      [last_From:]
      [last_To:]
      [last_Call-ID:]
      CSeq: 1 MESSAGE
      Content-Length: 0

    ]]>
  </send>
</scenario>
```

C.3 sipp/advanced_scenario/menu_select_yes.xml

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2  <!DOCTYPE scenario SYSTEM "sipp.dtd">
3  <scenario name="Submitting user input: yes">
4    <send>
5      <![CDATA[
6      MESSAGE sip:100@[remote_ip]:[remote_port] SIP/2.0
7      Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[
           branch]
8      Max-Forwards: 70
9      From: 420432100000 <sip:420432100000@[local_ip]:[
           local_port]>;tag=[pid]SIPpTag00[call_number]
10     To: 100 <sip:100@[remote_ip]:[remote_port]>
11     Call-ID: [call_id]
12     CSeq: 1 MESSAGE
13     Content-Type: text/plain
14     Content-Length: [len]
15     X-OMNI-User: 420432100000
16     X-OMNI-Organization: VladaCompany
17     X-OMNI-Service: VladaOutbound
18
19     1]]>
20   </send>
21
22   <recv response="200">
23   </recv>
24 </scenario>
```

C.4 sipp/advanced_scenario/menu_select_no.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "sipp.dtd">
<scenario name="Submitting user input: no">
  <send>
    <![CDATA[
    MESSAGE sip:100@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[
        branch]
    Max-Forwards: 70
    From: 420432100000 <sip:420432100000@[local_ip]:[
        local_port]>;tag=[pid]SIPpTag00[call_number]
    To: 100 <sip:100@[remote_ip]:[remote_port]>
    Call-ID: [call_id]
    CSeq: 1 MESSAGE
    Content-Type: text/plain
    Content-Length: [len]
    X-OMNI-User: 420432100000
    X-OMNI-Organization: VladaCompany
    X-OMNI-Service: VladaOutbound

    0]]>
  </send>

  <recv response="200">
  </recv>
</scenario>
```

C.5 sipp/advanced_scenario/send_reinit.xml

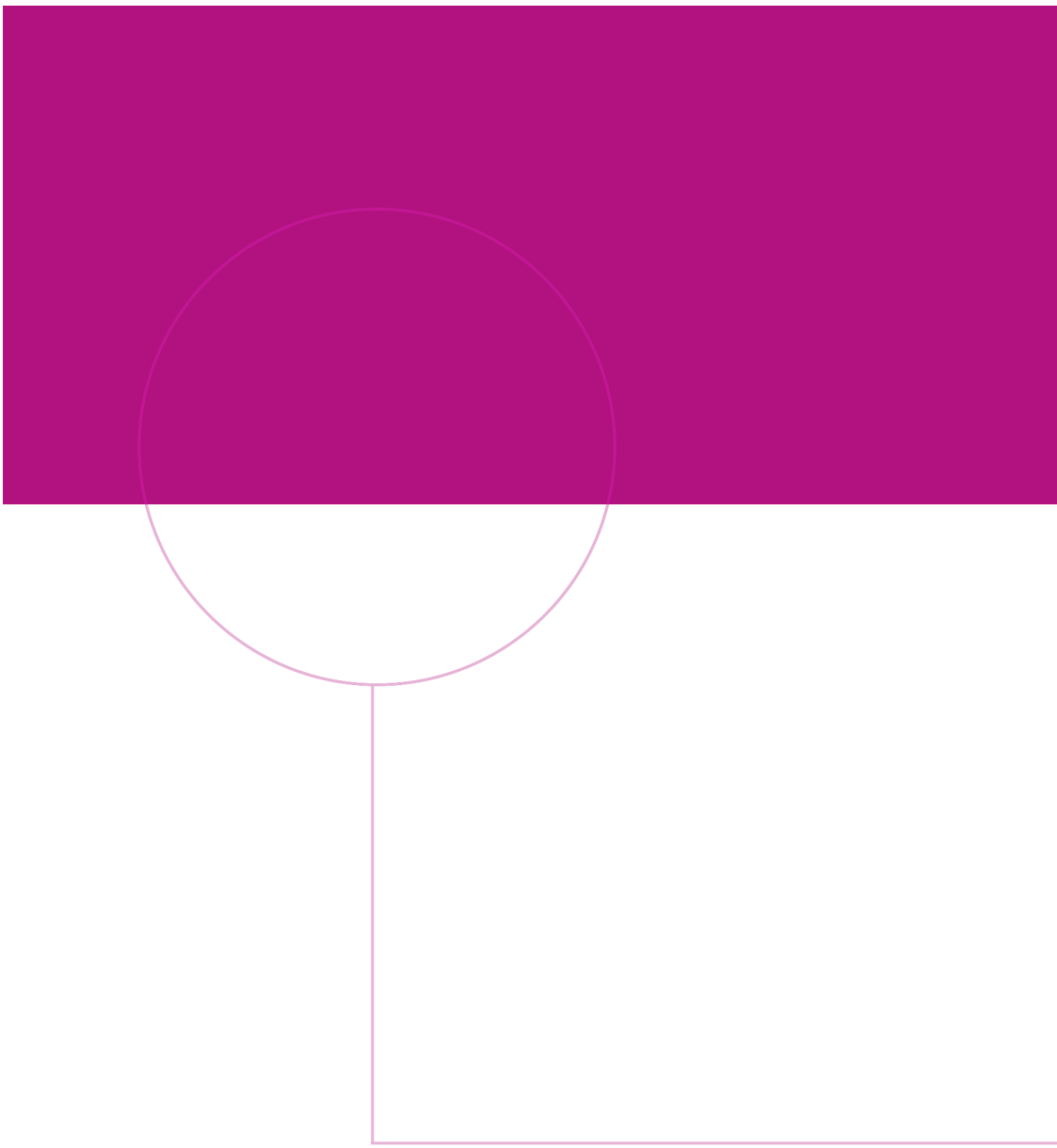
```
<scenario name="Resend request for service">
  <send>
    <![CDATA[
    MESSAGE sip:100@[remote_ip]:[remote_port] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[
        branch]
    Max-Forwards: 70
    From: 420432100000 <sip:420432100000@[local_ip]:[
        local_port]>;tag=[pid]SIPpTag00[call_number]
    To: 100 <sip:100@[remote_ip]:[remote_port]>
    Call-ID: [call_id]
    CSeq: 1 MESSAGE
    Content-Type: text/plain
    Content-Length: [len]
    X-OMNI-User: 420432100000
    X-OMNI-Organization: VladaCompany
    X-OMNI-Service: VladaOutbound

    Hello again!
     ]]>
  </send>

  <recv response="200">
  </recv>

</scenario>
```

C.6 sipp/advanced_scenario/send_nack.xml

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE scenario SYSTEM "sipp.dtd">

<scenario name="Simulate reception error">

  <recv request="MESSAGE">
  </recv>

  <send>
    <![CDATA[

      SIP/2.0 400 Bad request
      [last_Via:]
      [last_From:]
      [last_To:]
      [last_Call-ID:]
      CSeq: 1 MESSAGE
      Content-Length: 0

    ]]>
  </send>
</scenario>
```

C.7 sipp/run_advanced_scenario.sh

```
1  #!/bin/sh
2  ./sipp -sf advanced_scenario/send_init.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
3  ./sipp -sf advanced_scenario/send_ack.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
4  ./sipp -sf advanced_scenario/menu_select_yes.xml -t u1 -m 1
      -i 192.168.2.1 -p 5060 192.168.2.2:6060
5  ./sipp -sf advanced_scenario/send_ack.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
6  ./sipp -sf advanced_scenario/menu_select_no.xml -t u1 -m 1 -
       i 192.168.2.1 -p 5060 192.168.2.2:6060
7  ./sipp -sf advanced_scenario/send_ack.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
8  ./sipp -sf advanced_scenario/send_reinit.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
9  ./sipp -sf advanced_scenario/send_ack.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
10 ./sipp -sf advanced_scenario/menu_select_yes.xml -t u1 -m 1
      -i 192.168.2.1 -p 5060 192.168.2.2:6060
11 ./sipp -sf advanced_scenario/send_ack.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
12 ./sipp -sf advanced_scenario/menu_select_no.xml -t u1 -m 1 -
       i 192.168.2.1 -p 5060 192.168.2.2:6060
13 ./sipp -sf advanced_scenario/send_nack.xml -t u1 -m 1 -i
       192.168.2.1 -p 5060 192.168.2.2:6060
```



NTNU

Norwegian University of
Science and Technology