UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

Dynamic Adaptation of Service-Based Systems: a Design for Adaptation Framework

Martina De Sanctis

Advisor

Dott. Marco Pistore

Fondazione Bruno Kessler

Co-Advisor

Dott. Antonio Bucchiarone

Fondazione Bruno Kessler

February 2018

Abstract

In the Next Generation Internet landscape, different but interconnected metaphors, such as the Internet of Services, the Internet of Things and the Internet of People, to name a few, are living together in the same ecosystem. This led to an increase of the complexity of the contexts in which modern service-based systems must operate. Indeed, these systems need to cope with *open* and *continuously evolving* environments. They are expected to operate under dynamic circumstances, where dynamism is given by changes in the operational environment, changes in the availability of resources and variations in their behavior, changes in users goals, etc. In addition, modern systems consists of autonomous and heterogeneous components that, anyhow, must cooperate in a transparent way to accomplish the system goals.

In this settings, many approaches for self-adaptive systems providing their related methodologies and tools for the design, development and execution of systems in dynamic environments have been proposed. Adaptation, indeed, is considered a promising solution, and is still widely investigated, to release systems able to adapt and re-configure themselves to satisfy the changing conditions in a context-aware manner.

Unfortunately, the existing approaches tend to foresee the system adaptation needs and their related solutions at design-time. In this way, even if the adaptive behavior is effectively executed at runtime, the set of possible situations in which the system might need it, has been defined at design-time. Despite their effectiveness when applied in closed environments, current approaches are inadequate for the application in the typical open environments of the Next Generation Internet landscape. To deal with resources that constantly join/leave the system together with their provided functionalities, the traditional approaches require for continuous

involvement of IT and domain experts for the re-configuration of the system to accommodate it to the changes. This is notably an error-prone and time-consuming task. To increase the resilience of service-based systems to frequent changes, a new way of approaching the systems modeling and adaptation is needed.

In this direction, we claim that adaptivity is to be considered an intrinsic characteristic of systems rather than an exception to be handled. Differently from systems where traditional change detection and adaptation mechanisms can be used, the Next Generation Internet requires systems that are *adaptive "by design"*.

In this dissertation, we propose a novel *design for adaptation* approach of modern service-based systems that (i) allows the designer to model the system environment by abstracting from the concrete services (and their behaviors) that operate in it; (ii) provides a methodology for the *uniform* modeling of autonomous and heterogeneous services that is applicable in dynamic environments; (iii) provides methods for defining abstract adaptation requirements facilitating the dynamic services interoperability on top of a shared environment, by abstracting from their concrete implementation; (iv) supports the system execution via run-time and context-aware adaptation by exploiting advanced planning techniques for the dynamic and incremental service composition; (v) provides a *complete life-cycle* for the continuous development and deployment of adaptive service-based systems with a huge degree of *flexibility* and *extensibility*, to handle with open and dynamic environments.

The central aspect of the approach lies on how the used model allows us to overcome the services heterogeneity, on the one side, and to enhance the easy integration, reuse and interoperability of their offered functionalities, on the other side. The way in which services are linked to the context model allows services relations to be easily established and new services to easily join the system, thus expanding the context. Relations, which are set up by the exchange among services of offered/required functionalities, are also the way through which services can span their knowledge on the system environment (from a local to a global view). Consequently, with our approach it is possible to handle at run-time the dynamism of both systems and environments in a completely automated way.

The core enablers of a comprehensive framework defined over the proposed approach, have been implemented and evaluated on a real-world scenario in the smart mobility domain. Promising evaluation results demonstrate their practical applicability.

In addition, Collective Adaptive Systems (CAS) are spreading in new emerging contexts, such as the shared economy trend. Modern systems are expected to handle a multitude of heterogeneous components that cooperate to accomplish collective tasks. In this settings, a first extension of our framework in the direction of CAS has also been realized and evaluated.

Keywords
Next Generation Internet, Design for Adaptation, Incremental Service Composition, Collective Adaptive Systems, Smart Mobility

Executive Summary

A key challenge posed by the Next Generation Internet landscape, is that modern service-based systems need to cope with open and continuously evolving environments and to operate under dynamic circumstances. Dynamism is given by changes in the operational context, changes in the availability of resources and variations in their behavior, changes in users goals, etc. Indeed, dynamically discover, select and compose the appropriate services in open and expanding domains is a challenging task. Many approaches for self-adaptive systems have been proposed in the last decades. Unfortunately, although they support run-time adaptation, current approaches tend to foresee the system adaptation requirements and their related solutions at design-time. This makes them inadequate for the application in open environments, where components constantly join/leave the system, since they require for continuous involvement of IT and domain experts for the systems re-configuration. We claim that a new way of approaching the adaptation of systems is needed.

In this dissertation, we propose a *novel design for adaptation framework* for modeling and executing modern service-based systems. The idea of the approach consists in defining the complete life-cycle for the *continuous development* and deployment of service-based systems, by facilitating (i) the *continuous integration of new services* that can easily join the systems, and (ii) the systems operation under *dynamic circumstances*, to face the *openness* and *dynamicity* of the environment.

Furthermore, Collective Adaptive Systems (CAS) are spreading in new emerging contexts, such as the shared economy trend. Modern systems are expected to handle a multitude of heterogeneous components that cooperate to accomplish collective tasks. In this settings, an extension of our framework in the direction of CAS has also been defined.

The core enablers of the proposed framework have been implemented and evaluated in real-world scenarios in the mobility domain. Promising evaluation results demonstrate their practical applicability.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The purpose of this thesis is that of presenting a design for adaptation approach for the modeling and execution of service-based systems operating in dynamic environments. Anyhow, to understand the reasons that led to an urgent need for novel approaches supporting and enabling the adaptation of modern service-based systems starting from their design, we need to analyze the context in which these systems live. Thus, in section 1.1, we discuss the current operational context of modern service-based systems. Then, we highlight both the problem and the research challenges that we intend to address in section 1.2. With this knowledge in mind, an overview on the contribution of this thesis work is introduced in section 1.3. We close the chapter with the complete thesis outline given in section 1.4.

1.1 Thesis Context

The Internet of Services (IoS) is widespread and it is becoming more and more pervasive, since the trend is to deliver *everything as a service* [1], from applications to infrastructures, passing through platforms [2]. Furthermore, the scenario is still evolving: the IoS is envisioned as one of the founding pillars of the Next Generation Internet [3], together with new metaphors, such as those of the Internet of Things (IoT) and the Internet of People (IoP) [4].

In the IoS landscape, Service-Oriented Architecture (SOA) plays a central role. It is essentially a software engineering paradigm according to

which software is designed and developed in form of *interoperable services*. Indeed, one of the main potentialities of SOA is represented by *service composition*, that gives the possibility to create new services and *service-based applications* (i.e., composite services), by combining existing services (i.e., component services), to release new and most complex functionalities addressing specific goals. *Service-based systems*, instead, constitute the overall execution context where service-based applications run, in order to achieve their goals. They include the underlying services, software and hardware platforms, monitor and adaptation mechanisms, and so on. However, the scenario in which service-based systems must live and operate is influenced by different factors. Service-based systems must now face the increased *flexibility* and *dynamism* offered by modern service-based environments. The number and the quality of available services is continuously increasing and improving. In addition to new services, existing services may change their behavior, including their offered functionalities, and they may also join and leave the systems at any time. This makes service-based environments *open* and *highly dynamic*. In other words, service–oriented computing take place in an "open world" [5].

From an IoT perspective, the availability of devices and tools to access software-based services has also increased dramatically: services run on many of the devices that are used every day in a variety of contexts (e.g., smartphones, wearable, watches, smart home devices, connected health devices, smart farming devices). In this setting it is of relevant importance to take into account the different technical features of all kinds of devices (e.g., capacity, operating systems, interfaces), since they are becoming part of the infrastructure for publishing, discovering and executing services.

From the IoP side, instead, nowadays users are increasingly proactive and demanding. They participate in the systems they use and they can act as service providers as well. A suitable example comes from the sharing economy [1] trend that led to the spread of shared services. For instance, in the mobility domain, shared mobility services are based on the shared use of vehicles, bicycles, or other means. Users can act both as consumers (e.g., requiring a bike, a ride-share, etc) and as providers (e.g., offering their bikes, ride sharing, etc).

[1] https://en.wikipedia.org/wiki/Sharing_economy

In this settings, we believe that, in the Next Generation Internet landscape, service-based systems must have a rightful place at the crossroads of IoS, IoT and IoP.

To date, service-based systems are employed in a multitude of application domains, such as e-health, smart-homes, e-learning, education, smart mobility and many others. A particularly suitable domain to show the challenges of open and dynamic environments is the *mobility domain*. It is also very relevant both at European and world level. For instance, many past and running projects have been funded with the aim of providing solutions to support and stimulate a more integrated and sustainable mobility (e.g., [6, 7, 8]). Indeed, the offer of mobility services is quite large and heterogeneous. These services may differ depending on diverse aspects, such as the *service type* (e.g., journey planners, shared mobility services, traditional mobility services, online ticket payment services), the *offered functionalities* (e.g., user profiling, ticket booking, ticket payment, seat reservation), the *targeted users* (e.g., citizens, tourists), the *provider* (e.g., public vs. private), the *geographical applicability scope* (from local to global services), *availability constraints* (e.g., free vs. pay), etc. In addition, a journey organization, execution and monitoring consists of a set of different mandatory and/or optional phases (e.g., registration, login, planning, booking, check-in) that must be carried out, according to the involved real-world services. Indeed, different services can be exploited to perform the same functionality, and the pertinence of their application depends on the specific context of execution.

In these settings, *Smart Cities* are becoming one of the main drivers in the eruption of the Next Generation Internet. The urgent need for a more efficient and sustainable society, together with the spread of ubiquitous communication networks, highly distributed wireless sensor technology, and intelligent management systems, makes the Smart City ecosystem and ideal ground for IoS, IoT and IoP. In this context, the role of service-oriented architecture is to enable the integration and interplay between public and private city services to solve current and future challenges and support the creation and delivery of innovative and efficient services for the citizens [9].

Focus of the thesis. From here on, the focus of this thesis work will be more on the IoS landscape. However, the approach presented in this dissertation has been designed with the IoT and IoP paradigm in mind. Indeed, further extensions in their direction are part of our near future work.

In last decades, the aim of service oriented computing has been that of encouraging the creation and delivery of services, also taking advantage of the support of Web. Besides, automated service composition is a powerful technique within service-oriented computing allowing to compose and reuse the existing services as building blocks for new services (applications) with higher-level functionalities, based on users' needs. Additionally, the role played by companies and organizations is also considerable. They are publicly providing their services to allow third-party developers to exploit them in defining new services, thus enhancing their accessibility [10]. This is of relevant importance in the IoS scenario, since it implies the availability of reliable services offering functionalities that cannot be easily implemented by single developers or small companies (e.g., Google Maps, Paypal, Rome2Rio). Different organizations are building on this trend to provide online platforms for the management of well-defined RESTful APIs through which these services can be accessed. For instance, ProgrammableWeb [2] now has more than 10,000 API in its directory. As a consequence, both researchers and practitioners are highly motivated in defining solutions allowing the development of service-based systems, by exploiting existing available services.

Yet, as we already mentioned, the continuously increasing number and quality of available services, naturally makes service-based environments *open* and *highly dynamic*. This strongly demand *self-adaptive* service-based systems, that is, systems able to both *adapt* to their actual context (i.e., the currently available services) and *react* when facing new context situations (e.g., missing services, newly available services, changed services, changes in the user requirements and needs).

However, "*there are still major obstacles that hinder the development and potential realization of service computing in the real world*". This is clearly stated in [10], where a number of experts in service computing pro-

[2] http://www.programmableweb.com

vide a manifesto about a ten-years roadmap guiding the service oriented computing in overcoming its current limitations and addressing new emergent challenges. In fact, the evolution of the Future Internet as well as the latest Next Generation Internet vision further challenge the IoS paradigm. As discussed in [11], "service-oriented computing has to face the ultra large scale and heterogeneity of the Future Internet, which are orders of magnitude higher than those of today's service-oriented systems". Indeed, the main limitation in the literature is that, in our opinion, the huge number of service-based methodologies and approaches are limited by the fact of dealing with one or a few of the features required by adaptive service-oriented systems to operate in modern environments.

For instance, *microservices* [12, 13] have been proposed to support the development of distributed applications, as the evolution of monoliths, quite more compliant with modern service storage and provision platforms, such as the cloud. However, microservices focus more on the evolution of systems either than on their adaptive behavior. *Software product lines* (SPL) (e.g., [14, 15, 16]) or *mashup applications* (e.g., [17]) focus both on supporting the variability of service-oriented applications when executed in dynamic environments, providing a way for dynamically combining the most appropriate services for the current situation. However, SPL approaches are meant for closed environments, where the available services are known a priori. Mashup applications, instead, do not provide a generic modeling approach for defining adaptive services.

Self-adaptation is still one of the main concerns in the context of IoS. Many approaches have been proposed with the aim of increasing the flexibility of applications to support both their *adaptation* and *evolution* needs.

They span from *rule-based approaches* (e.g., [18]), to *artifact-centric approaches* (e.g., [19]), passing through *dynamic software product lines* [20] or *configurable process modeling* approaches (e.g., [21, 22]), to name a few. However, the adaptation of service-based systems should not pre-define adaptation rules as part of the systems' logic and, then, trigger them when specific needs arise. Indeed, this would prevent an effective management of the dynamicity of the environment that might also cause adaptation

needs that have not been pre-defined at design-time. Besides, most of the approaches falling in these categories are mainly limited by their low applicability in open environments and their distance from real execution contexts, often preventing them from leaving the realm of research.

Lastly, the level of complexity of modern systems is progressively increasing. They actually form socio-technical systems, composed of distributed entities (e.g., services and service providers, software and human participants) interacting with and within the environment. In this context, multiple participants must adapt their behavior in concert to respond to critical run-time impediments. This trend is bringing to the spreading of *Collective Adaptive Systems*(CAS), where the adaptation is a feature of the collectiveness, thus demanding for collective adaptation approaches [23].

1.2 Problem and Research Challenges

Despite the progress reached in service oriented computing, a further effort is still required to shape dynamic and context dependent services and applications that can efficiently and effectively operate in modern environments.

> A key challenge that still needs to be overcome, so that the creation of innovative services becomes a reality, is the capability of dealing with the **continuously changing and complex environments** in which applications operate.

To acknowledge this claim, an important consideration must be made about the before-mentioned *openness* and *dynamicity* of service-oriented environments that we will particularly stress in this dissertation.

Open environments are quite challenging. As we said, they are characterized by services entering and leaving the system at any time. Besides, systems can be affected by the unavailability or malfunctioning of services and from changes in their procedures, which could be affected even by changes in regulations and norms. This means that the services on which service-based systems rely on might not be known a priori and/or not available at execution time. In addition, if we consider the amount of publicly available services, it is easy to notice that *different* services might

implement the *same* functionality, even if with different procedures, and each of them may be more or less appropriate in a specific execution context. Because of this, to perform accurately, service-based systems must be aware of the specific execution environment during their execution, thus operating differently for different contextual situations, by exploiting the effectively available services.

Thus, what we want to emphasize is that the openness of the environment further increases its dinamicity. This is why the traditional change detection and adaptation mechanisms meant to be used in dynamic but closed environments, are not suitable for the application in modern open environments.

In other words, differently from applications where traditional adaptation mechanisms can be used, the Internet of Services requires systems that are **adaptive by design**.

Consider for instance the case of a *Travel Assistant* system, supporting service users (tourists, citizens) and providers (transportation means companies, municipalities, mobility service providers) in their daily operation and management of mobility services (e.g., buses, trains, bike-sharing, ride-sharing) and journeys, within a smart city as well as among different cities/countries. The implementation of such a system requires to deal with a variety of heterogeneous services: from generic services (e.g., access management) to domain specific services (e.g., journey planning); from legacy city services (e.g. access to traffic information or local parking availability from local systems) to new innovative services (e.g., a travel assistant supporting users for the whole travel duration); from fundamental infrastructure services (e.g., wireless sensor network connecting sensors and smart devices) to application-like and end-user interaction services (e.g., mobile and Web apps for users). Given the variety and autonomous nature of these services, changes are not only frequent, they are an inner characteristics of the system.

In this context, the requirements that a service-based system must fulfill to contemporary deal with all the characteristics of modern environments can be summarized as follows. The system must be capable to propose complex and personalized solutions (*customizability*) taking advantages of

the variety of services (*interoperability*), while taking into account the state of the environment (*context awareness*). During its execution, they must be able to react and adapt to changes in the environment that might occur and affect its operations (*adaptivity*) and the system must be open and extensible for new services to become part of it (*openness awareness*). In addition to these, there are commonsensical requirements strictly related to the nature of services. The system must, indeed, consider the heterogeneity of the services (*heterogeneity awareness*) and their autonomous nature (*autonomy awareness*), while providing up to date and reliable information and solutions (*information accuracy*). Lastly, what is expected from modern systems is that they are deployable in different environments without an ad-hoc reconfiguration from the developers (*portability*). This particularly calls for general and *domain independent* approaches that can be easily fitted to different domains.

These emergent requirements strongly affect the *design*, the *execution* and the *adaptation* of service-based systems. With these premises, we come out with two main research challenges (RC) that we intend to address with the work presented in this dissertation, which are:

RC1: provide a solution for modeling and executing adaptive service-based systems that is able to handle the above-mentioned requirements as a whole, rather than only a subset of them;

RC2: provide a solution that can be easily applied in real-world service execution contexts, thus able to be used also out of the realm of research.

1.3 Contribution

These premises motivated the work presented in this dissertation about a novel *design for adaptation approach of service-based systems*. The approach is based on the idea that adaptation cannot be considered an exception to be handled, but instead, service-based systems must be *adaptive by design*.

To address the issues illustrated in the previous section, we agree with and we have been inspired by the suggestion given in [24], where the au-

thors argue that mechanisms enabling adaptation should be introduced in the life-cycle of systems, both in the *design* and in the *run-time* phases. To achieve this, the adaptation must be supported by a *coherent design approach* supporting both the *definition* and the *application* of adaptation.

In very general terms, the idea of the approach consists in defining the complete life-cycle for the *continuous development* and deployment of service-based systems, by facilitating (1) the *continuous integration of new services* that can easily join the systems, and (2) the systems operation under *dynamic circumstances*, to face the *openness* and *dynamicity* of the environment.

This has been realized by specifying a new design approach allowing the separation between the application and adaptation logic of each modeled service and, thus, of the entire service-based system. This is particularly reached by using two separate but correlated models: (1) the *Domain Model* describing the operational environment of the system. It allows designers to abstract the *domain concepts* of the referred domain. (2) The *Domain Objects Model* allowing developers to uniformly specify the autonomous and heterogeneous services in the environment, and their *dynamic interaction*, as the concrete and diverse implementations of the domain concepts specified by the domain model. Thus, different implementations (i.e., services) of the same concept can be interchanged as needed, depending on the execution context. In addition, the approach allows for the specification of services behaviors (i.e., domain objects core processes) and functionalities (i.e., domain objects fragments) as *dynamically customizable processes* that can be concretely specified at run-time, to guarantee the *context-aware execution* of systems. To make this processes customizable and adaptable, they are labeled with *composition requirements* predicating over the domain model. These annotations guarantee the detachment of adaptation requirements from concrete services specification. To further increase the flexibility of the system, in order to handle the high dynamicity of the execution environment, the approach allows service-based systems to dynamically span their knowledge on the whole domain, at execution time. In this way, systems can expand their partial view on the domain, by discovering available services on the need.

At this point, what is missing is the explanation of how the adaptation (dynamic process customization) is performed. As we said, because of the openness and dynamicity of the environments, the adaptation of service-based systems should not be pre-defined, since this would prevent an effective management of the environment dynamicity.

To this aim, our approach provides different *adaptation mechanisms and strategies* allowing systems to adapt to different situations (e.g., select the proper services, react to a context change, etc). In particular, to guarantee as much as possible the success of the provided systems (e.g., in terms of their applicability, correct execution, coherent adaptation, etc), their adaptation must be performed closer to their execution, that is, when the context is known. To achieve this, the approach exploits advanced techniques for *dynamic and incremental service composition and re-configuration* [25], allowing to effectively deal with changes occurring at different levels in the system (e.g. entrance and exit of services, change in provided functionalities, change in system requirements). Indeed, the exploited adaptation mechanisms and strategies are particularly suitable in open environments.

Lastly, to face the increasing level of complexity of modern systems, which is flowing into collective systems, the approach has been extended to model CAS and perform collective adaptation. The applicability in real-world execution environment is, finally, demonstrated by the development of a *Travel Assistant* in the mobility domain that completely relies on real mobility services, previously integrated in the system, by defining and connecting them in terms of the presented design for adaptation approach.

Figure 1.1 summarizes the approach by abstracting its fundamental features, showing how it works to deal with both *individual* and *collective* adaptive systems.

Publications.

Some of the contributions making this thesis work are also reported in the following published papers on which this thesis is founded. A preliminary version of our approach is illustrated in [26] (*ESOCC 2014*). A further extension and revision is reported in [27] (*ICSOC 2015*). Here we introduce the basic constructs, techniques and enablers of the design for adaptation

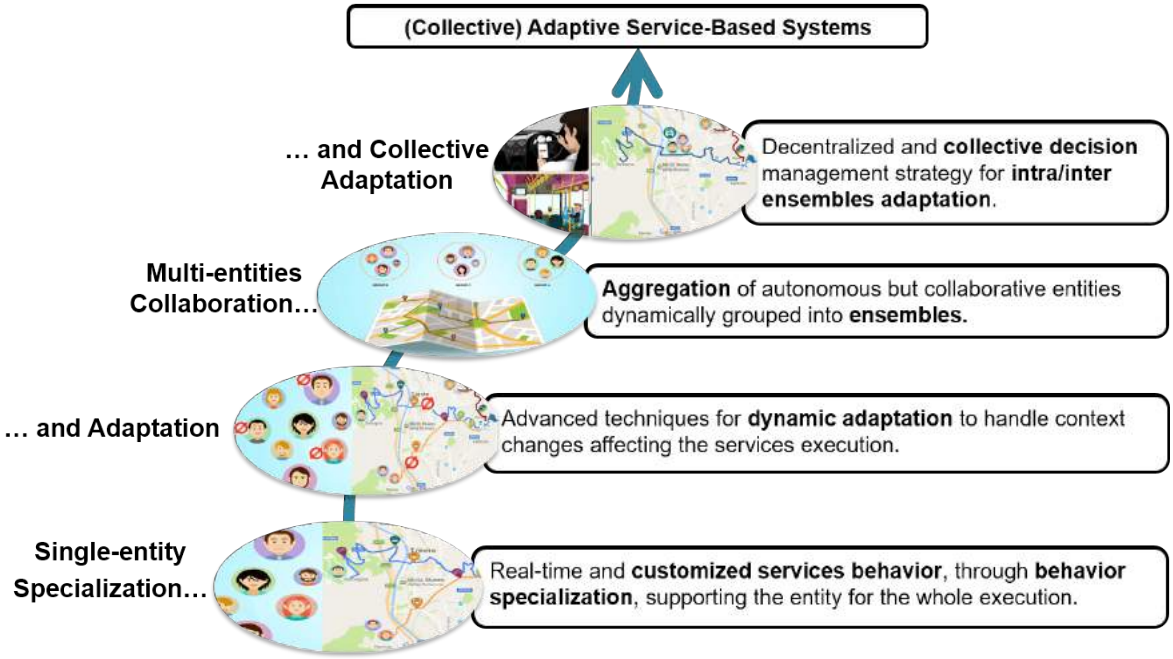


Figure 1.1: From Individual to Collective Adaptive Systems.

solution. In [28] (*ICWS 2016*) the approach has been further formalized and preliminary evaluation results are given. The development of a comprehensive framework based on our approach and the implementation and evaluation of a service-based application developed within it has been presented in [29] (*ICSOC 2017*). Here we propose ATLAS – a world-wide travel assistant that enables the integration and interplay between new and real-world mobility services, dynamically operating and adapting in the mobility domain. The extension of the design for adaptation approach in the direction of CAS, together with the definition of a collective adaptation algorithm for their execution are reported in [30] (*ASOCA at ICSOC 2016*). The implementation and evaluation of the extended approach for CAS, instead, is given in [31] (*Best Demo Award at the demo track at ICSOC 2016*). Here we provide a simulator for collective adaptive systems. It currently implements and executes a *Urban Mobility System* scenario. In particular, it shows both its normal execution, via ensembles formation, and its collective adaptation when facing adaptation needs affecting one or several ensembles. As concern CAS, we further proposed a Domain Specific Language (DSL) for engineering CAS in [32] (*eCAS at SASO 2017*)

and its related tool in [33] (*demo track at SASO 2017*). Lastly, a preliminary investigation about service co-evolution within our framework has been presented in [34] (*WESOA at ICSOC 2015*).

The development of our approach was partially performed in the scope of the ALLOW Ensembles European project [35] that adopted part of the approach, specifically the one relating to the modeling and management of CAS.

1.4 Thesis Outline

This thesis work is organized in eight chapters. Chapter 2 provides an overview on background information on the main topics of the thesis. We give some basic information on SOC and its related standards and we review the state of the art in the areas of the design and the dynamic adaptation of service-based systems. The chapter also identifies some important open issues in these areas. Chapter 3 provides an overview of our design for adaptation framework for modeling and executing service-based systems. It introduces the motivating example of this dissertation, which is about a smart travel assistant in the mobility domain, that helps us to discuss the motivations of our research. It also introduces some basic concepts that are useful to understand the subsequent chapters. In chapter 4 we introduce our approach for the design of service-based systems relying on the interoperability of different services and supporting the definition of adaptation in the systems models. Chapter 5, instead, details the adaptation mechanisms and strategies allowing systems to operate in open and dynamic environments and to adapt their behavior to the operational context at run-time. In chapter 6, we introduce Collective Adaptive Systems, by also providing a relative literature overview. Then, we explain how the design for adaptation approach of chapter 4 can been extended with specific constructs to model CAS. Furthermore, we illustrate how the adaptation mechanisms discussed in chapter 5 are exploited for the normal execution of CAS, allowing different entities to be grouped into ensembles and to collaborate. For the decentralized and collective decision management required by CAS, instead, we further provide a collective adaptation

approach, by illustrating its corresponding algorithm and evaluating it on two different scenarios. Chapter 7 is devoted to the implementation and evaluation of a comprehensive framework implementing all the enablers required by the approach presented in this thesis. In particular, both the original approach and its extended version for CAS are shown in action in two different demonstrators realized within the framework, namely ATLAS [3] and DeMOCAS [4] that are freely accessible as open-source tools. Finally, chapter 8 contains a final discussion on the requirements coverage provided by our solution. In addition, we discuss a number of works according to the future directions that we intend to investigate in the near future. Eventually, each chapter is equipped with a final section where we briefly discuss the content of the chapter, its possible open points and its connections with the rest of the work.

[3] ATLAS is available at: `https://github.com/das-fbk/ATLAS-Personalized-Travel-Assistant`. A video on the execution of ATLAS is available at: `https://vimeo.com/231387418`. [4] DeMOCAS is available at: `https://github.com/das-fbk/DeMOCAS`. A video on the execution of DeMOCAS is available at: `https://www.youtube.com/watch?v=H0_LjptwZDg&feature=youtu.be`.

Chapter 2

State of the Art

In this chapter we discuss the most significant advances in those areas of service-oriented computing that relate to the management of service-oriented systems and applications. As starting point, in section 2.1 we provide an overview of the principles of service-oriented architecture and we introduce important definitions by describing the main standards currently in use. The remaining sections of the chapter are devoted to the current progress in two main areas of interest that are central to this dissertation. In particular, in section 2.2 we survey some of the most relevant works dealing with the modeling of service-based systems and applications. We focus especially on those modeling approaches whose aim is, among the others, to increase and support the flexibility and dynamicity of the modeled systems. Section 2.3, instead, surveys relevant works concerning methodologies for the adaptation of service-based systems and applications.

To organize this literature review, we have decided to survey approaches belonging to those areas with which we have been asked to compare our solution during the years of this PhD work. In addition, we provide a detailed literature review about decision-making and run-time adaptation in Multi Agent Systems in chapter 6, which is devoted to the extension of our design for adaptation approach to deal with Collective Adaptive Systems. We conclude the chapter in section 2.4 by briefly discussing the identified problems with the existing approaches. We finally give some hints on how the work presented in this thesis is supposed to improve and extend the state of the art in the considered areas of interest.

2.1 Overview on Service-Oriented Architecture

Service-Oriented Computing(SOC) and *Service-Oriented Architecture* (SOA) are the wide scope which this thesis work belongs to. In [36], SOC is summarized as follows:

> Service-oriented computing represents a new generation distributed computing platform. As such, it encompasses many things, including its own design paradigm and design principles, design pattern catalogs, pattern languages, a distinct architectural model, and related concepts, technologies, and frameworks.

From this definition, it arises the complexity of the SOC paradigm with all the elements making it. Among them, SOA represents the next step in the evolution of distributed computing. It is essentially based on the idea to construct business applications of *reusable* components called *services*. To effectively and efficiently exploit the potentiality of SOA, however, a set of *design principles* [36] have been defined. Despite the numerous proposals for services modeling languages, these design principles are in any case the foundation that must be guaranteed by every language. Thus, we briefly summarize them in the following. Within the same service inventory, services use the same service description standards. In other words, they all adhere to a service contract specifically defined by service description documents (*standardized service contract*). Service contracts are decoupled from their surrounding environment, especially from their concrete implementation. In such a way, they can evolve independently from service consumers and from their implementation (*service loose coupling*). Service contracts publish only essential information about services, by hiding their internal logic (*service abstraction*). Moreover, the service logic is defined in such a way so that it can be reused entirely or in some of its parts (*service reusability*). Going on, services exercise a high level of control over their underlying run-time execution environment (*service autonomy*) and, as concerns the management of their state data, they defer it when possible, to minimize the resource consumption (*service statelessness*). Services should facilitate their discoverability. To this aim, they are

equipped with communicative meta data through which they can be discovered and interpreted (*service discoverability*). Last but not least, services are composable elements, independently from the composition complexity (*service composability*). Follow these principles means to realize *flexible* service-based applications, by significantly decreasing their development cost and further maintenance support.

In the last few decades, a multitude of languages for modeling services as well as approaches for their efficient composition have been proposed. In the rest of this section, we give an overview on the standard languages and approaches that constitute the basis of all the further extensions that have followed over the years.

2.1.1 Services Modeling Languages

Despite SOA is based on different open standards and can be realized by using different technologies, such as REST [37] to name one, its most popular implementation is based on *Web Services* [38]. Even if our research is not strictly associated with Web services but, to the contrary, it is independent of any specific service-based technology, we often use Web services-related terminology [39]. Moreover, since Web services represent the most complete implementation of SOA, we give here an overview on this standard technology.

According to the World Wide Web Consortium (W3C), *a Web service is a software system supporting the machine-to-machine interaction over a network. It is equipped with an interface described in WSDL and it interacts with other systems as prescribed by its description, by using SOAP messages. Messages are transmitted using HTTP with an XML serialization, by also exploiting other Web-related standards* [1].

There exists a set of standards concerning the Web services technology and supporting the basic infrastructure required to implement Web services. We describe here the most relevant by summarizing their detailed specification given in [38]. The standard for the Web service message format is *Simple Object Access Protocol* (SOAP) [2], which uses XML as data format. The central role played by SOAP is that of communication pro-

[1] `https://www.w3.org/TR/ws-gloss/` [2] `https://www.w3.org/TR/#tr_SOAP`

tocol, whose aim is that of providing a standard way to encode different protocols and interaction mechanisms into XML documents that can be easily exchanged through the Internet [38]. In particular, SOAP has been designed to be stateless and one-way, thus supporting the service loose coupling principle mention above. Then, the Web service infrastructure provides unified description documents to declare how a Web service can be exploited. The most important ones refer to *service interface* and *service protocol*. A service interface is a set of supported operations, each of them with its *input* and *output messages*. The *Web Service Description Language* (WSDL) [3] is a standard language for Web service interfaces specification based on XML. WSDL allows designer to specify the programming interface of a Web service through the definition of its supported methods with their input and output messages. Besides supporting the standardized service contract principle, the definition of WSDL has been also required by the need to cope with the lack of a centralized middleware managing the message transport, which has arisen with the evolution from conventional middleware to Web services infrastructure [38]. A service protocol, instead, gives the valid operations sequence and it describes how the service state is affected by the execution of these operations. Service protocols can be specified with any workflow languages. The most famous one is, probably, *Abstract BPEL* that is part of the *Web Service Business Process Execution Language* (WS-BPEL) [4], a language for executable service-based business processes. Lastly, a quite relevant standard allowing for the *advertisement* and *discoverability* of services is the *Universal Description Discovery and Integration* (UDDI) [5] specification. A standard way to publish and locate services was strongly required to use Web services in a pervasive manner, thus leading to the standardization of the Web services registry provided by UDDI. Indeed, UDDI standardizes registries through which services can advertise themselves on the Web. In turn, customers can use the registries to discover, locate and execute services. In particular, UDDI defines both data structures and APIs for publishing service descriptions in the registry as well as for querying it [38]. The goal of UDDI is twofold: it supports the service discovery by the developers and, more important, it enables the

[3] `https://www.w3.org/TR/#tr_WSDL`
[4] `http://docs.oasis-open.org/wsbpel/2.0/OS/`
[5] `https://www.oasis-open.org/standards#uddiv3.0.2`

dynamic binding among services.

In summary, from these basic standards we can derive the conventional model of service-oriented architecture, based on Web services, that works as follows. Services are developed by service providers and equipped with service description documents, such as WSDL. These services are then submitted to a UDDI registry that makes them available on the Web. Service consumers can discover available services by using the registry and, thus, obtaining their descriptions. At this point, service consumers can interact with the selected services via SOAP messages.

Web services, of course, embrace many more aspects than those covered by SOAP, WSDL, BPEL and UDDI. Obviously, several extensions to this conventional model have been performed from its initial definition. Extensions have been made necessary for different purposes. Probably the most relevant refers to the management of a coordinated execution of multiple services. Indeed, besides the one-to-one interaction between a service and its consumer, what is quite challenging still today is the communication and collaboration management among different service providers and service consumers, operating to achieve certain goals. In fact, one of the potential of Web services is that they can be easily combined in such a way that simple services of different programs can interact and be composed to deliver *value-added services*. This is also due to the fact that Web services define the interactions among components, without any restrictions on the implementation technologies or platforms. Because of this, *service composition* has become a major challenge in SOC.

2.1.2 Service Composition

Service composition is probably the major potential offered by SOA. It essentially gives the possibility to create new services and *service-based applications* (i.e., composite services), by combining existing services (i.e., component services), to release new and most complex functionalities addressing business goals. Although it is apparently a simple task, service composition is quite difficult and it also span among different research areas. Indeed, it involves the specification of *composition requirements*, the *discovery* and *selection* of suitable services and their further composition.

Moreover, after the composition has been performed, it might also be monitored and adapted.

From the arising of Web services infrastructure, service composition was a natural way for managing the complexity of defining complex services, as the combination of the functionalities provided by other Web services. It is easy to notice how service composition exploits both the service composability and the service abstraction design principles, by allowing the definition of composite services built on top of services at a lower level of abstraction. Naturally, there exist several approaches for the service composition proposed in the last decades [40], and a lot has been also done, from the beginning, to make this process *automatic*. Indeed, with the evolution of the Internet and the spreading of the Internet of Services, researchers started soon to propose and realize approaches able to, firstly, make bindings between services more flexible and, secondly, to automatize as much as possible the service composition process, to make services and service-based applications faster, dynamic and adaptive. Some of the early work done in this direction are [41, 42, 43]. Briefly, in [41], the authors aimed at overcoming the typical point–to–point delivery of services, by proposing a platform, so-called eFlow, for specifying, enacting, and monitoring composite services defined as business processes. [42] probably represents one of the first attempt to arrange the automatic e-Service composition. Its implementation involves the so-called composition synthesis referring to the synthesis of a new composite e-Service, by specifying the coordination among the component e-Services exploited to obtain the composite one. The work in [43], instead, goes beyond the simple reuse of composite services, by pushing the reuse of the compositions themselves as orchestrators able to flexible bind *different* component services, instead of specific ones. This work clearly builds on top, and further refines, the concept of service binding, aiming at postponing it as later as possible in the composition life-cycle, to make it flexible and even open to the exploitation of diverse dynamically selected services. However, a complete review on this topic goes a bit out of the scope of this dissertation. In fact, even if our proposed design for adaptation approach exploits service composition techniques for the run-time adaptation of service-based systems, we do not provide any new service composition approach. The focus of

our solution is much more on the design side of service-based systems, in such a way of making them *adaptable by design*, by potentially exploiting different available service composition solutions.

As a consequence, while for the interested reader we strongly suggest [40] as an overview on Web services composition approaches, in the rest of this section we mention the two conventional ways to compose services, under which almost all approaches fall down. We refer to *orchestration* and *choreography* [44]. Orchestration is distinguished by the presence of an *orchestrator*, that is, a central component that fully control the composition logic and simplifies the interoperability among component services. Choreography, instead, gives interaction protocols among different services participating in the composition in a decentralized fashion. As a consequence, with choreographies the composition logic is distributed among all component services. Furthermore, these two approaches are not mutually exclusive. To the contrary, they can also be used in combination.

Standard languages have been defined for specifying orchestration and choreographies. Among all, we mention Business Process Execution Language (BPEL), as the reference standard for services orchestration. In particular, Executable BPEL is the part of BPEL devoted to the description of executable service-based business processes. It is used to define work-flows, by specifying for instance service invocations activities, send and receive message activities (i.e., for asynchronous interaction), events, operations and variables. These activities are, then, organized through a set of standard control-flow constructions (e.g., loops, parallel and sequential execution). Different extensions of BPEL have also been proposed to overcome some of its limitations. An example is WS-BPEL Extension for People (BPEL4People) [45] that introduced human activities to BPEL in order to allow it to define general purpose business processes rather that orchestrations of Web services.

Web Services Choreography Description Language (WS-CDL) [6] is probably the main standard for specifying choreographies. It is used to describe peer-to-peer collaborations between parties, by defining their observable behavior (message exchange) from a global perspective. Collaborations specified in WS-CDL allow predefined business goal to be achieved.

[6] `https://www.w3.org/TR/ws-cdl-10/`

Lastly, Business Process Model And Notation (BPMN) [7] is also a powerful approach for the definition of business processes on conceptual level. It is widely used for specifying service-based business processes and it is exploited for defining both orchestration and choreographies.

To conclude this high-level overview on service composition, we find relevant to mention that, according to researchers and practitioners, service composition approaches can follow essentially two different strategies, namely *top-down* and *bottom-up* [46]. In particular, these strategies refer to the first phase of the service composition life-cycle, known as *synthesis*. Top-down approaches firstly foresee the composition at a higher level of abstraction, for instance by defining an abstract process model. Then, the predefined abstraction is converted into an executable composition through the discovery and selection of suitable services and their further composition. Bottom-up approaches derive the service composition logic based on services definitions and a specific abstract goal to be reached. Furthermore, automated service composition allows for the automatic combination of services, by using an automated reasoner as, for instance, in AI planning based service composition approaches [47].

As concerns the design for adaptation approach presented in this dissertation, we already said that it exploits automated planning techniques for the dynamic and incremental service composition [25], for the context-aware customization of processes and the run-time adaptation of systems. In particular, based on what we said about service composition, [25] can be classified as a bottom-up approach able to perform indistinctly both orchestrations and choreographies. Moreover, the approach in [25] performs particularly well in dynamic environments.

In conclusion, as regards the standard approaches of service composition, such as those of orchestration and choreography, they have some crucial limitations. A major problem of these approaches is that most of them are based on the assumption that during the composition requirements specification, the application designer knows the services to be composed. Besides, some of them, such as WS-CDL and BPEL, remain focused on the syntax level without considering the semantic aspects of composition, which are, instead, necessary in context-based applications. Others

[7] `http://www.bpmn.org/`

standard approaches, such as OWL Web Ontology Language for Services (OWL-S) [8] and Web Service Modeling Ontology (WSMO) [9], to name a few, have introduced the management of semantic knowledge in their models to drive the services' composition and interoperation but, despite this, they do not allow processes to be defined at run-time, through dynamic service composition.

2.2 Design of Service-Based Systems

This section is devoted to a discussion about the design of service-based systems. It is carried out by reviewing some of the design approaches that are closer to or have some in common with the solution that we present in this dissertation.

Considering the advances in service-oriented computing in the last decades and the definition of the Web Services technology with its further standardization, the development of service-based systems has become increasingly interesting. Indeed, service-based systems allow for combining numerous services into business applications implementing a business logic, also complex, that is provided by the collaborations among the involved services. Moreover, defining service-based systems allows the fully exploitation of the potentiality of Web Services and the generation of a new potential, as arising from their collaborations.

A considerable amount of approaches for the development of service-based systems have been proposed. As a consequence, in [48] the authors have highlighted the importance of defining a sound *service-oriented design and development methodology* to deal with contemporary complex and heterogeneous computing environments. By reviewing methods and techniques used in different methodologies, they come out with a "*life-cycle of the web services development that is of crucial importance to specify, construct, refine and customize highly volatile business processes from internally and externally available Web services*", which is currently relevant today. The implementation of SOA is a complex task that involves different aspects such as networking, artificial intelligence, security, perfor-

[8] `https://www.w3.org/Submission/2004/07/` [9] `https://www.w3.org/Submission/WSMO/`

mance, data management and others [49], requiring for a wide range of technologies, tools and skills. From a web services life-cycle development perspective, instead, the focus should be mainly on designing a service-oriented architecture able to be in line with business process interactions between trading partners sharing a common goal (e.g., trading of a product). The proposed service-oriented design and development methodology is based on the iterative Web services development life-cycle depicted in Figure 2.1. The methodology applies well to both Web services and business processes. In particular, it focuses on business processes as reusable building blocks and its objective is that of achieving services integration and interoperability.

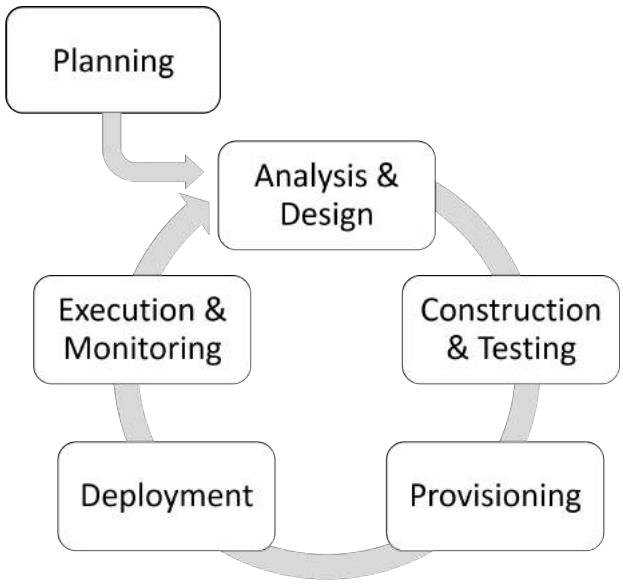


Figure 2.1: Service-oriented design and development methodology [49].

An extended version of this methodology can be also found in [50]. Briefly, the involved phases are:

- the *Planning* phase to identify the characteristics and the feasibility of service solutions in a given context, by defining business needs as goals;
- the *Analysis* phase to investigate the requirements of a new application;

- the *Design* phase refers to the design of useful business processes on top of existing or new defined services. During this phase, service-oriented design principles must be followed (e.g., service coupling and cohesion above all) and the service composability must be guaranteed;
- the *Construction & Testing* phases, include the Web services implementation (e.g., from scratch or by composing reusable Web services), the description of their interface, their behavior and their validation against the requirements;
- the *Provisioning* phase involves both technical and business aspects and it supports the service client activity;
- the *Deployment* phase during which the service provider publishes the service's interface and its implementation definitions;
- the *Execution & Monitoring* phases during which the service is fully operational and the quality of services of systems and applications released as service-oriented solutions is measured and monitored.

The ultimate purpose is that of designing and developing enterprise solutions as composite applications involving different services connected via well-specified contracts allowing to define and automate business processes.

From an abstract perspective, the approach highlighted in Figure 2.1 always applies to the process of constructing service-based systems of loosely-coupled, autonomous and reusable services, and it contributes to support the adaptive capabilities of services, strongly needed today. Obviously, different approaches can implement it in different ways. For instance, each phase can be more or less automatic. However, up today new dynamic and advanced models and technologies must be provided and exploited for implementing this methodology, in order to release service-based adaptive systems able to operate in *open* and *dynamic* environments, also in the situation in which the exploitable services are not foreseen at design-time.

In the rest of this section, we survey existing solutions for the modeling of services that are relevant in the area of the provision of service-based systems. Recent advances in service oriented computing and *cloud computing* [51], in particular with regard to *microservices* [12, 13], and *open*

services, are providing new opportunities to make significant progress in developing service-based systems allowing designers to deal with and solve complex real-world challenges.

In recent years, the **cloud computing** metaphor has emerged with the aim, among others, of providing advantages over traditional in-house IT services, such as an elastic storage and computing resources over the Internet, through the application of a pay-per-use model. Even if the cloud computing is quite recent, probably first introduced by Eric Schmidt in 2006 [52], it will probably play an important role in the development of the future Internet of Services. Moreover, it already represents a research area itself, with dedicated conferences, workshops, journals etc. Thus, in this dissertation we focus especially on those challenges in the cloud computing paradigm that are relevant from an Internet of Services perspective.

In cloud computing environments, the role of services is even more crucial, since everything can be released as a service, from applications to infrastructure. Indeed, Figure 2.2 shows the cloud computing model enabling the future IoS, as reported in [2]. It represents the different provisioning models that can be offered in a cloud environment. The *Software as a Service* (SaaS) model allows for on-demand access to applications. The *Platform as a Service* (PaaS) model allows for providing platforms on top of which one can develop and deliver services and applications. The *Infrastructure as a Service* (IaaS) model allows for the provision of elastic computing infrastructures (servers, network, storage, etc). Although this representation is made from a high-level perspective and cloud environments are not necessarily organized as such, the need for efficient approaches for services development and management, is quite evident. The authors in [2] envision the importance of IaaS as the foundation of the future IoS and give a list of relevant challenges that should be addressed for meeting the IoS requirements. We agree with their envisioning, especially as concerns the growing need for computing infrastructure and resources required by modern service-based systems. This is not a new challenge, as shown for instance in [53], where autonomic computing techniques have been applied to automatically solve the resource provisioning problem for distributed process execution engine. However, it still remains an open problem that can be more effectively addressed by exploiting the potential

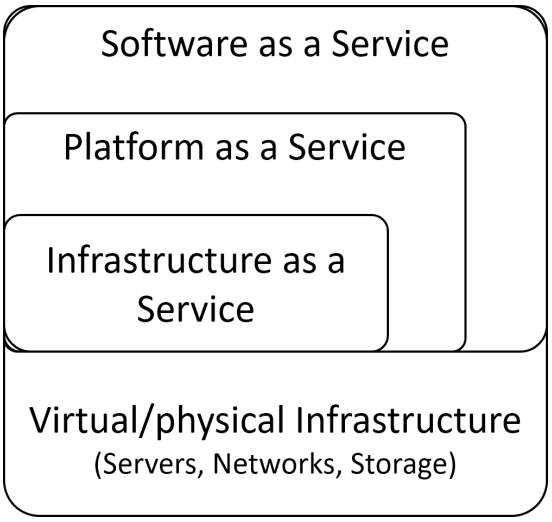


Figure 2.2: The cloud computing model [2].

of modern IaaS and their ability to provision elastic computing resources. However, in the context of this dissertation, we particularly focused on a subset of the IoS requirements and we consider them at the PaaS model level. Indeed, the design for adaptation framework that we are going to present in this dissertation can be potentially viewed as a PaaS, since it might act as a service delivery platform, on top of which adaptive service-based applications can be defined. The authors particularly highlight the need for supporting the dynamic (context-aware) service provision, their resiliency and reliability, among other challenges. As refer to the *support for the dynamic service provision*, in [2] the authors claim that to provide efficient service virtualization, cloud platforms should decouple the service interface from the service implementation. In this way, services could be dinamically mapped on the different resources. The *context-awareness* of applications and services, instead, is becoming more and more challenging with the increase in relevance of the information in a given context. Just think about the wide use of mobile applications, social networks, pervasive computing, as context-aware applications adapting their behavior to the surrounding environment. What is still an open challenges in supporting the provision of context-aware applications is the ability to effectively allow these applications to collect, analyze and exploit context information. Lastly, among the challenges listed in [2], one clearly refers to the

availability of services offered by the Internet. This strongly requires for *reliable* and *resilient* service-based applications. In the design for adaptation framework discussed in this dissertation, these two aspects are supported by the fact that composition requirements are abstracted from the concrete services so that, when applications need to satisfy them, they can look among the currently available services able to accomplish the specific requirements.

However, also according to the International Data Corporation (IDC) [10], cloud computing has several issues. We particularly mention a lack of standardization, a lack of customization, and limited interoperability. Currently, a standard way to interface with a cloud does not exist, and each provider exposes its own APIs. The risk is that, even if the cloud is probably the first enabler of the *everything as a service* paradigm, it might emphasize the problem of the heterogeneity among services, by further adding the heterogeneity of the cloud providers of these services. Moreover, this would also affect the interoperability among services, which is also still an open challenge.

In [54] the authors propose an architectural level solution to overcome the current limitations of cloud computing, with respect to customization and standardization, particularly. The authors claim that the cloud architecture is of crucial importance when moving into cloud computing. This architecture should support specific capabilities for migrating traditional services and applications to cloud computing systems, while also supporting all kind of involved users (e.g., cloud vendors, developers, customers). However, the traditional cloud computing multi-layer architecture, such as that in Figure 2.2, acts as an obstacle, especially because each **aaS* has its own features, requirements and output, making difficult for vendors to customize services. That's why the authors propose a dynamic and customizable architecture, called *Template as a Service*, that provides, in particular, a single service layer allowing the interaction with the cloud resources and services of the different **aaS* layers. Each cloud vendor can define a front-end of several templates where each template gives cloud services to the end-users.

In the context of this dissertation, we can not but argue about **mi-**

[10] `https://www.idc.com/`

croservices and the **microservices architectural style** [12]. From when microservices started gaining popularity, many publications have been made on them, since they inspired numerous research work. Here, we focus mainly in giving an overview on the microservices paradigm, its relevant features and what has led to the evolution of distributed architectures into the microservices architectural style.

In [55], in their survey titled *Microservices: yesterday, today, and tomorrow* the authors present an interesting overview on the history of software architecture, spanning from the diffusion of services, until microservices. While the debate on the role of microservices over SOA is still open (the question is if they represent an *evolution* or an *implementation* of SOA [56]), the fact on which everyone seems to agree is the basic motivation that led to the spread of microservices. This refers to the need for moving from monolith applications to distributed applications. "*A monolithic software application is a software application composed of modules that are not independent from the application to which they belong*" [55]. It is quite clear that, in the current context of cloud-based distributed systems, monolithic applications represent an obstacle, due to their difficult to distribute. In [55] the authors also draw up a list of the most relevant issues related to monoliths, that we sum up in the following:

- difficulty to maintain and evolve;
- problems with the addition or updating of libraries, which easily results in inconsistency of the system;
- rebooting of the whole application is required after any single module's change;
- the deployment of monoliths requires for a compromise about the final configuration, because of conflicting requirements on resources of the constituent modules;
- the scalability of a monolith application is limited by its own structure;
- monoliths represent a technology lock-in for developers.

Due to these and other issues, related to the evolution of technologies, programming languages, development approaches and so on, the microservices architectural style has been proposed as a potential solution to overcome the limitations of monoliths. In [55], the following definition are given.

> A *microservice* is a minimal independent process interacting via messages. A *microservice architecture* is a distributed application where all its modules are microservices.

What is important to highlight is that microservices are independent components, each implementing only one or a few functionalities, deployed in isolation. The different microservices in a system are, then, coordinated and composed via messages.

A deeper view on microservices is given in [13]. The author firstly exposes his point of view, according to which microservices emerged as a trend from real-world use. Then, he describes the main features and key benefits of microservices. The most relevant are the following: microservices are *small* fine-grained services, with precise boundaries; they are *autonomous*, living as separate entities exposing an API for the communications with other entities. Among the key benefits of using microservices with respect to monolithic applications, we particularly highlight (i) the *resilience*, since they avoid a failure to cascade; (ii) the *scalability*, since they avoid to scale everything as a piece; (iii) the *easy of deployment*, since each microservice can be deployed independently from the rest of the system; and (iv) the *composability*, since they support the reuse of functionalities that can be composed and consumed in different ways. In [13] the author also emphasize that the SOA principles [36] that microservices stress more are *loose coupling* and *high cohesion*. Indeed, he states that: "*When services are loosely coupled, a change to one service should not require a change to another. The whole point of a microservice is being able to make a change to one service and deploy it, without needing to change any other part of the system.*" At the same time, he also accentuate the importance of high cohesion, by saying that: "*We want related behavior to sit together, and unrelated behavior to sit elsewhere. So we want to find boundaries within our problem domain that help ensure that related behav-*

ior is in one place, and that communicate with other boundaries as loosely as possible." While defining our design for adaptation approach we also took into account the loose coupling and high cohesion design principles. In particular, the loose coupling is guaranteed by the *Domain Object* artifact, since each domain object in our model represents an autonomous and independent service whose dependencies with other domain objects are dynamically established only on the need. The high cohesion, instead, is given especially by the definition of the *Domain Model* as an abstraction of the operational environment of the system. It allows designers to abstract the domain concepts of the referred domain. Then, different services in the environment can provide concrete and diverse implementations of the same domain concept, if they provide a service whose behavior matches with the abstract behavior provided by the domain concept.

Although there is a lot of enthusiasm around microservices, as they represent the novelty in the development of service-based systems, it is necessary and important to understand when they are effectively useful and when they are not. To this aim, we have found very interesting the interview reported in [57], where the authors question microservices experts and insiders, such as Mike Amundsen, James Lewis, and Nicolai Josuttis, about their experiences in using microservices. While they essentially consider microservices as a best practice approach for realizing SOA, they also underline the need for applying different architectural approaches for each set of functional and cross-functional requirements. In other words, their opinion is that microservices should be used if the system that must be developed really needs it. From their experience, the interviewees suggest to use microservices in order to enable business goals, thus keeping grounded the implementation in business value. Moreover, what we find an important lesson, is the suggestion made by Nicolai Josuttis, when he says that the distribution always has a price to pay. Indeed, more flexibility means also more complexity for the system maintenance. Thus, it is important to do not switch to distribution if it is not really needed. Lastly, what we strongly share with Mike Amundsen and we find relevant for our research work are the two main points that he underlines when referring to the refactoring of a system. In particular they refer to the importance of (i) decoupling the interfaces from the internal models, and

(ii) re-sizing the system components. This is the right way for establishing standards and practices for defining and implementing interfaces between service provider components and service consumers. As we will see later on in this dissertation, these two points are of relevant importance also in our design for adaptation approach. They mainly relate to the use of *fragments* acting as the interface exposed by the system components. In our approach, fragments implements both the decoupling between interfaces and internal models and, at the same time, their use facilitate the decomposition of big components into smaller ones, each exposing a part of the whole component's interface via a subset of fragments.

An interesting work on microservices is presented in [58]. Here the authors focus on how to support the *automatic deployment of microservices*, given their dynamic nature. In particular, they propose JRO – Jolie Redeployment Optimizer that is a tool for the automatic and optimized deployment of microservices written in the Jolie language [59, 60]. In fact, before detailing the work in [58], we find relevant to review Jolie too. Jolie is an open-source programming language [11] for the development of distributed applications based on microservices, which uses both computation and composition primitives. In Jolie, (micro)services are the building blocks of the language and they are characterized for being distributable and reusable by design. Services communicate by exchanging messages and, more important, the language is protocol agnostic. Indeed, not only it supports different protocols but it also provide an API for the development of new ones, if required. Jolie also supports different ways for building complex service-based software systems, such as orchestration, aggregation, redirection and embedding. For concurrency purposes, multiple instances of the behavior of a service, modeled as processes, can run. Processes can directly communicate via message correlation.

Going back to [58], for its execution JRO relies on three main components: (1) *Zephyrus* [61], a tool allowing for the automatic generation of a detailed architecture, by starting from an abstract description of a target application; (2) *Jolie Enterprise*, a distributed framework for deploying and managing microservices written in the Jolie language; and, (3) *Jolie Reconfiguration Coordinator*, a tool that, by interacting with Zephyrus,

[11] `http://www.jolie-lang.org/`

produce the optimized deployment planning, given a desired configuration and context.

In conclusion, what clearly emerges from microservices and their architectural style is that they offer a sound architecture for *scalability* and *evolvability*. Indeed, and this is probably a current limitation of this approach, microservices focuses more on the evolution of systems either than on their adaptive behavior [62].

From our research perspective, we see different commonalities between our design for adaptation approach and microservices. For instance, we mention the implementation of the loose coupling principle and the support for the definition of little-size interoperable components. However, as opposed to microservices, our approach also support the run-time adaptation of service-based applications. This is due to the possibility of defining adaptive by design services that further exploit techniques allowing them to dynamically adapt their behavior to the surrounding environment. What we find interesting is to deeper investigate if and how we can combine the two approaches for the development of service-based systems, in order to exploit the potentialities of both of them. A preliminary investigation in this direction is reported in [63]. Here the authors define a road-map made of key research objectives towards the utilization of domain objects as a model for the microservice architecture. The authors plan to reach each objective by following an evidence-based strategy, through the implementation of a concrete case study in the IoT domain. In particular, the question they want to reply is the following: "*is the domain objects formalism suited to describe software system to be built according to the microservices architecture? In other words, is the formalism over-expressive or under-expressive?*". This is certainly one of the future directions that we will investigate in consideration of future extensions and improvements of our approach.

After talking about service-oriented computing, cloud services and microservices we can not but speaking about DevOps [64]. Indeed, DevOps emerged as a natural consequence of the last decades evolution of service oriented computing and it is, in such a way, sustained by the cloud and microservices paradigms. DevOps is a paradigm, or better a practice, whose aim is that of guaranteeing a rapid and efficient value delivery to market.

It strongly promotes a tight collaboration between the developers – the *Dev* – and the teams managing the deploy and operation the systems – the *Ops*. DevOps is focused on decreasing the gap between the design of a product and its operation. To this aim, it introduces design and development practices and approaches to the operation domain and vice versa. This is quite clearly shaped by the DevOps application lifecycle, which is always used to introduce it and which is reported in Figure 2.3. What has

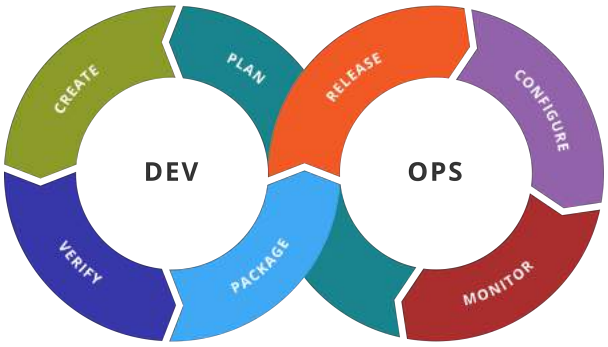


Figure 2.3: The DevOps Application Lifecycle.

to be clarified about the DevOps lifecycle is that, while it provides the DevOps process to be followed, it is not constrained to any specific software development paradigms, tools, languages etc. To the contrary, the lifecycle is mainly based on a set of pillars, such as collaborative development, accelerated deploy, continuous collaboration and feedback, continuous validation, continuous integration, and it can be implemented in any way a DevOps team decides. Obviously, there exist many practices or tools particularly suitable for the realization of the DevOps approach, since they mostly help in bringing the gap between development and operations, but their use is not mandatory at all. From our research point of view and considering the idea behind the design for adaptation framework presented in this dissertation, the application of the DevOps approach is certainly one of the future developments to apply to our approach, in order to, above all, make the most of all the potential and improvements that the DevOps implementation would certainly bring.

Service-oriented computing shares several characteristics with **software product lines** (SPL). In the classic view of software product lines (SPL) the designer analyzes a software family as a whole and establishes the com-

mon and reusable assets that form its basic platform, as well as the possible application-specific customizations [14]. A feature model [15] specifies the alternative variations and constraints that can exist for each feature.

The integration of services and features have already been investigated in the literature [65]. In the model proposed in [65], features are operationalized through atomic or composite services. This model builds on a two-phase lifecycle that firstly integrates services and features and then, in its second phase, uses the integrated model to derive a product that satisfies the end-users desired feature selections.

In [66] different types of development of SPL are distinguished (e.g., proactive, reactive and extractive). In [16], instead, the authors propose a *Generative* SPL development based on variability-aware design patterns. The authors started from the consideration that design patterns are already used to handle variability in the implementation of SPL. However, they claim that there is not a specific and standard method for the proactive development of SPL exploiting design patterns for defining variable functionality. They propose their generative SPL development where role models are exploited. Role modeling allows developers to describe the dynamic collaborations among objects, instead of their design. In [16], role models are used to link design patterns to a variability model, by capturing their relation.

Although SPL are not a recent approach, they are currently widely used, even in different application domains, as demonstrated by [67] and [68], for example. Moreover, it continuously evolves, for instance by adhering to new methodologies, such as the Agile development, as in [69]. In particular, in [67] SPL have been exploited to develop a End User Development (EUD) tool, that is a framework enabling and supporting end users to create software applications for smart spaces.

In [68], instead, SPL have been applied in the robotics field. In particular, in this work the authors integrate the cloud robotics and SPL techniques. While the former deal with low-level aspects, such as algorithms, processors and robots, SPL are used as support for the end users to deploy and configure complex robotics applications.

Lastly, in [69], Agile practices are integrated into SPL with the aim, among others, of balancing agility and formalism. Used in combination,

they allow certain goals, such as improving reuse and productivity, reducing time to market and decreasing development costs, to be achieved faster. This last sentence, summarizes the aim and the potentialities of SPL. However, they do not deal with the management of the run-time adaptation of the systems they model. This is probably the main reason why they evolved into *dynamic* SPL, which we survey in the next section.

Considering the context of this dissertation, we find of current relevance the role played by **Open services**. Open services are services that are easy to understand and to access and that can be exploited to develop applications or can be improved to provide new value-added services on top of them. When launching and managing a new service, many service providers choose to do all the work themselves. They are unaware that there are service providers available to help their tasks, deploying, launching and managing their service infrastructure and ecosystem and, at the same time, to manage the access to these new services.

Web APIs are the most common way to specify such services. At the same time, the combination of semantic technology and Web Services (i.e., semantic web services (SWS)) increases the automation degree of automation in tasks such as the discovery, composition and mediation of services. To overcome the limitations of SWSs (i.e., the use of non-standard and unfamiliar languages for description) a model for Linked Open Services has been introduced in [70] in which services are viewed primarily as RDF "prosumers". With the rise in popularity of web APIs, platforms for their management and customization, called *API management platform*, have been provided. These APIs can be combined to build composite applications and value-added services. However, while advances in web services and their composition enable automation and reuse, new productivity challenges have emerged in the case of APIs. The service developer requires sound understanding of the different service types and access-methods, as well as being able to format input data, or parse and interpret output data in the various formats [71] (e.g., XML, JSON, SOAP, HTTP).

ServiceBase [72] proposes a Unified Services Representation Model, and a "programming" knowledge-base, where common service-related low-level logic can be abstracted, organized, and reused by other applications developer. With ServiceBase a set of APIs have been implemented that expose

a common and high-level interface to developers for integrating services in a simplified manner, despite their heterogeneity. Finally, many "smart cities" initiatives performed or currently executing across Europe are now working on open platforms, collaborative web tools and interoperable services. Some examples are: IES Cities [12], Open Cities [13], and FIWARE [14].

Web services and APIs are widely adopted by programmers to build new applications in various programming languages on top of open data, Internet of Things and crowd and cloud services. Organizations like Mashery [15] and Apigee [16] are building on these trends to provide platforms for the management of APIs. For instance, ProgrammableWeb [17] now has more than 10,000 API in its directory. These services can be combined to build composite applications and higher-level services using composition techniques. However, while advances in SOA, in terms of Web service description and services composition have enabled automation and reuse, complete solutions for open services management are yet required. The open services management requires for complete solutions to support the development and distribution of new services but there is still a need to make services easy to understand and to access for third parties.

In the next section, we focus on approaches devoted to guarantee a certain level of self-adaptation of service-based systems developed through their application.

2.3 Dynamic Adaptation of Service-Based Systems

In last years, the increasingly growing dynamicity and openness of the environments in which service-based systems live has posed new challenges as regards their development and evolution over time. For instance, *adaptivity*, *scalability* and *software reuse* are strictly crucial for applications to keep up with their environment, but they are not yet completely and efficiently handled. Thus, the need for adequate development approaches of service-based systems has emerged.

Self-adaptation is one of the main concerns in the context of the Internet of Services. It has been studied in different areas, such as Self-Managed

[12] http://iescities.eu [13] http://opencities.net/ [14] https://www.fiware.org/
[15] http://www.mashery.com [16] http://apigee.com [17] http://www.programmableweb.com

Systems [73], Multi-Agent Systems [74], Ubiquitous Computing [75, 76] and Service-Oriented Computing [77, 78]. Kramer and Magee proposed a famous three layer architecture model [73] supporting the assertion that, to be self-managed, a system should reconfigure itself to satisfy the changed specification and/or environment. In [74], instead, the authors propose Multi-Agent Systems (MAS) to support the engineering of self-adaptive systems. We provide a detailed literature review about decision-making and run-time adaptation in MAS in chapter 6, which is devoted to the extension of our design for adaptation approach to deal with Collective Adaptive Systems.

In the area of Ubiquitous Computing (UC), we mention [76] that proposes a task-based self-adaptation infrastructure for supporting the automatic (re)configuration of UC environments. Another approach for self-adaptation in UC is proposed in [75]. It allows the selection of an appropriate plan, from a library, that can be used to achieve a user goal, by taking in consideration her preferences and available resources. The approach also supports the definition of user preferences per goal requests, by deriving preferences at run-time based on the executed tasks context. However, although the approach uses a component to monitor the execution, diagnosis and context-aware adaptation mechanisms are not specified.

The need for self-adaptive software also triggered the development of self-adaptive architectures. The work presented in [79], for instance, represents a predecessor of all the research work done on both short-term adaptation and long-term adaptation, better known as *system adaptation* and *system evolution*, respectively. The authors predicted that advances bringing to self-adaptive software would come from a broad landscape of research areas specifying different techniques for the effective design and operation of self-adaptive systems. Moreover, they support "the dominant role of software architecture in planning, coordinating, monitoring, evaluating, and implementing seamless adaptation." In particular, what we want to highlight in the work presented in [79] is the following statement:

> While technical advances in narrow areas of adaptation technology provide some benefit, the greatest benefit will accrue by developing a comprehensive adaptation methodology that spans

adaptation-in-the-small to adaptation-in-the-large, and then develops the technology that supports the entire range of adaptations.

We claim that, with the design for adaptation presented in this dissertation, we have taken a lot of steps forward in the direction of a comprehensive framework supporting the definition of adaptive systems, from their design, to their evolution, passing through their operation via dynamic adaptation. Nevertheless, we are also aware that we are still not there, but we think we are on the right path.

From a service-oriented perspective, the need for self-adaptive applications led researchers to the definition of self-organizing service-oriented architectures. For instance, we mention [78], where the authors motivate and promote self-organizing SOA to overcome the practice of centralized components and services, and the need of manual tasks in the applications development. To this aim, the authors consider scalable and decentralized solutions, while investigating about the increase of automation in different phases of the whole life-cycle of service-oriented applications, such as, service discovery, composition, monitoring and so on. Eventually, they come out with a roadmap whose aim is that of addressing the definition of a decentralized, self-organizing, and light-weight infrastructure simplifying part of the management of the services life-cycle.

Obviously, also in the Software Engineering area self-adaptation is still a relevant topic, as emerges from [80], and a lot has already be done in the past decades. In [80], the authors pose their attention mostly on four essential topics of self-adaptation, namely, (i) the design space of adaptive solutions, (ii) processes, (iii) from centralized to decentralized control, and (iv) practical runtime verification and validation, as challenging and open to new research and investigations. In particular, as we will see in this dissertation, we believe that the design for adaptation approach that we propose advances the state of the art concerning three of the four identified topics in [80], such as the design space, processes and decentralized control.

An approach quite close to our, as concern the model definition and the exploitation of planning techniques, is presented in [81], where the **SAP Business Objects** model is defined. The main idea behind this work was

to support and automatize both the creation and adaptation of business processes in dynamic business environments. In order to use technologies from the planning field, as already done in this context, the authors have faced the problem of managing the additional modeling overhead caused by the planning, due to the high cost of designing the planning model. To overcome this limitation, the authors had the idea to exploit the results from model-based software development, by applying even the one-to-one reuse of models built for software engineering purpose. As a consequence, the Status and Action Management (SAM) model has came out. It is used both for software development and planning purpose, and it has been created in the context of the *SAP Business By Design paradigm*, whose aim is to develop flexible service-oriented IT infrastructure. It supports the design and management of changes by enclosing individual functionalities as software services, which can be easily combined. At the same time, it supports planning without imposing static workflows. The SAM planning functionality is integrated into the BPM modeling environment targeted at the creation of new processes, which is part of the SAP NetWeaver BPM. Infact, the goal of this work is to adapt the infrastructure to each SAP customer, to allow them to create their own processes, by flexible combining provided functionalities. However, the SAM model is not meant to be used at run-time, but it is thought for being used at design time.

Considering the need for adaptation of systems at the architecture level and including it as an inner characteristic of their design offers several potential benefits, such as: (i) an appropriate abstraction level to describe dynamic changes in the system, (ii) the possibility of having a scalable and dynamic execution environment that easily deals at run-time with different types of changes, and (iii) the generality that allows the design of solutions for a wide range of application domains [73, 24]. Following this principle, in this thesis work we have proposed an approach that, exploiting advanced service refinement and re-configuration techniques, supports the design, development, and operation of service-based systems that are resilient to a wide range of changes. Various other approaches have been proposed in this direction. In the following we consider those approaches that might be applied in scenarios similar to the Travel Assistant presented in this dissertation.

We will start analyzing **rule-based approaches**, since most design for adaptation approaches fall in this category. We mention MoDAR [18], a model-driven approach for the development of adaptive service-based systems. They aim to abstract the variable part of a business process simplifying the process definition by using rules to model its complex structure and to capture its variable behavior. Then, rules can be linked in specific cutting points of the base process. In [82], the authors tackled the problem of the unpredictable execution of service-based applications. They focus on how to evolve a running service composition proposing a way for modeling composite services as artifacts that can change at run-time by exploiting the *models@runtime* concept [83]. Here, rules are used to model adaptation needs (events) and adaptation actions, from the design-time phase. However, in [82] the software engineer intervention is required to manipulate the run-time model of services and, being the adaptation and application logic mixed, the model is not flexible. Moreover, rules are not suitable for managing continuous and unpredictable changes in *open* environments, since they need to be continuously revised.

We want to discuss a bit more in detail the just mentioned **models@runtime paradigm**. At run-time, service-based systems have to cope with many different situations and contingencies, which can not be analyzed and checked entirely at design-time. Managing the variety of different execution variants requires a specification of variation points at design-time and binding them at run-time depending on situation and context. A promising approach to managing complexity in run-time environments is to develop adaptation mechanisms that leverage software models, referred to as models@runtime [84]. Work on models@runtime seeks to extend the applicability of models produced in MDE approaches to the run-time environment. The idea of models@runtime is to leverage models both at design-time and at run-time for monitoring, dynamic adaptation and evolution of software systems. Models@runtime has been, and still is exploited by different approaches. We mention, for instance, the work in [85], where the authors aim at addressing the problem of managing unanticipated changes in dynamic environment, which may make the system unable to meet its quality requirements. The solution proposed consists in automatically evolving the system model to follow the evolution of quality

parameters over time. The system adaptation envisioned in this approach leverages on the models@runtime idea by using the model to determine the system run-time configuration.

Another category of approaches, quite close to our proposal, that emerged in recent years are **artifact-centric approaches**. In [19] the authors present a formal framework defining *Business Artifacts*. They represent conceptual entities made of their attributes and states, their tasks modeling the services performed on such artifacts, and business rules defined in ECA style specifying the life-cycle of an individual artifact, as well as the control logic of a process executed between interacting artifacts. In [86] instead, the proposed approach focuses more specifically on the specialization of artifact-centric processes applying both to the individual artifact's life-cycle and to the interactions between artifacts. The specialization is made starting by base process models and the behavior-consistency between specialized and base processes is assured. Although the work in [19] supports flexibility and re-usability since it models the behavior of each artifact through a set of ECA rules, it suffers from the same limitations of rule-based approaches described in previous paragraph. The approach in [86] allows processes among interacting artifacts to be specialized differently depending on the context, but this is not done dynamically during the run-time phase. Indeed, this work supports more the reuse of business processes rather than their dynamic refinement.

The adaptation of services shares several characteristics with **Dynamic Software Product Lines** (DSPL). As introduced in section 2.2, in SPL the designer analyzes a software family as a whole and establishes the common and reusable assets that form its basic platform, as well as the possible application-specific customizations [14]. Then, a feature model specifies the alternative variations and constraints that can exist for each feature. DSPL extend SPL to support late variability that allows adapting features with no down time and without violating their constraints. Moreover, the feature model itself becomes a run-time entity that can dynamically evolve to satisfy new variability dimensions in the system. In some work ([87], [88]), the authors exploit the concepts of dynamic software product lines (DSPL) [20] for the realization of techniques for developing adaptive service-based systems. In DSPL a software family is analysed as

a whole and both common and reusable assets are established, together with the possible customizations of the application. Then, feature models are used to specify alternative variations that can be used for adaptation. In [87] the authors bring forward the idea of their previous approach, called DAMASCo [89], which mainly consists in allowing services reuse in pervasive systems. In DAMASCo, services are accessed via their public interfaces and context-aware techniques are used for the service discovery, composition and adaptation. To overcome the limitations of DAMASCo, due to the unmanaged variability of services, the authors provide an extension based on both SOA and DSPL, where feature models are used to represent the variability of the services by modeling families of adaptable software products and to allow, as a consequence, the realization of dynamic service composition in a context-aware manner. In [88] the authors present LateVa whose underlying idea is quite similar to that in [87]. Similar processes containing common and variable parts are called process variants. A base model defines the common part while fragments defines concrete variable parts. Base models are annotated with variation points where fragments can be dynamically entered. In this way, both software reuse and run-time variability are addressed. Anyhow, the main limitation of DSPL-based approaches is that they assume a close world where process variants (and thus fragments) are pre-defined. For this reason, these approaches do not fit well in open environments in which system components, and their provided functionalities, can enter or leave the system in any moment.

We now move on evaluating some **configurable process modeling approaches**. To start, we mention the work in [90] that proposes a framework allowing for a fast and flexible modeling of business processes. Services are integrated in a plug-and-play manner in which activities are selected from a repository and then dropped into a process. However, in [90] from the design perspective there is not the idea to model adaptable by design services, but rather to speed up the design. Thus, from an adaptation viewpoint, only ad-hoc modifications can be managed.

In last years, configurable process modeling approaches are focusing on how to design processes in order to give them a degree of flexibility, aimed to support both adaptation and evolution needs. The concept of

configurable process model has been introduced by [91], and it denotes configurable models that are obtained by merging different variants of process models. In [21], the authors address the problem of making business processes more flexible in order to adapt to different needs and requirements. Considering the state of the art in the area of *configurable process models*, the authors aim is to overcome the limitations and issues related to traditional approaches in the field, in which the variability of processes is given by merging existing process models into configurable reference models, with the consequence of adding complexity to the design process. In contrast, the proposed approach provide an algorithm for merging *configurable process fragments* instead of entire configurable models. Process fragments are small process building blocks that can be composed together and replace an activity in the process. Dealing with fragments instead of models reduce the complexity and the computation time of the process configuration. However, this approach is thought for assisting the process designer by automatically provide fragments to be used in the variability points of a process, during the design phase. In [22] the authors also focus on the variability management of process models, in the context of Process-Aware Information Systems (PAISs) and, in particular, from the organizational perspective, meant as one of the views of a process model, together with the control and behavioral ones. In order to provide context-based process model variants, the authors implement two algorithms, both based on general concepts, such as abstraction and polymorphism allowing the process model variants generation to adapt to different scenarios coming from different situations. Also in this case, however, the goal is to help the designers and practitioners in managing variations of process models by avoiding redundancy and inconsistency.

Configurable Event-driven Process Chains (C-EPCs) represent the evolution of classical event-driven process chains (EPCs). In order to capture and manage the variability in EPC process models, C-EPCs identify a set of variation points in the model and constraints to limit the combinations of possible variants that can be used in the different variation points [91].

As one could imagine, in the Next Generation Internet era, the growing number of online resources, data and services led to the rise of methodologies and tools to create applications by the combination of these resources.

These applications are referred to as **mashups**.

> A *mashup*, in web development, is a web page, or web application, that uses content from more than one source to create a single new service displayed in a single graphical interface. The term implies easy, fast *integration*, frequently using API and data sources to produce enriched results that were not necessarily the original reason for producing the raw source data. The main characteristics of a mashup are *combination*, *visualization*, and *aggregation*. To be able to permanently access the data of other services, mashups are generally client applications or hosted online [18].

To date, there exist different classes of mashups and numerous tools and research works on them. Most existing tools and approaches focus on the composition of web-based interfaces and functionalities and they are addressed also to non-expert end-users, bringing mashups to become a popular web development paradigm. For the context of this dissertation, we focus on ***service mashups***. A service mashup is, simply, a value-added service build from existing services. In particular, we consider the works in [17], [92] and [93], which are closer to our work.

In [17], the authors focus on the importance of *context-awareness* and *adaptivity* of service mashups in dynamic environments. Indeed, service mashups are often defined and executed in the Web that is strongly characterized by a high dynamicity. The availability of services frequently changes by affecting mashups that might need to be reconfigured based on the execution context (e.g., reputation, organization, location). Thus, the authors propose semiautomatic reconfiguration and replaceability strategies supporting developers in the redesign of compositions of services behind the mashups. In particular, the proposed adaptation strategy relies on two main blocks: *capabilities* and *requirements*. The former describes non-functional service properties to determine a service's applicability in a specific mashup context. They describe those behavior properties that cannot be directly derived from the service interface (e.g. reconfigurability). The

[18] `https://en.wikipedia.org/wiki/Mashup_(web_application_hybrid)`

latter specifies the necessary capabilities for a particular mashup. They depend on the current context and define a desirable mashup configuration. Requirements are defined as event-driven rules. The performed adaptation process executes monitoring and adaptation phases. The mashup monitoring observes context changes and determines which mashup configurations are affected. The adaptation is triggered when capability constraints no longer match the current service configuration. The mashup adaptation takes into account the set of requirements and determines the set of services with the best fitting capabilities, by using utility function. Lastly, the mashup developer selects among these provided services and she reconfigures the mashup accordingly.

In [92] the authors propose a novel approach that combines the modeling power of software product line feature models with AI planning techniques to perform service mashups composition allowing non-expert users to both build and optimize service mashups. The approach relies on (i) the integration of services and features proposed in [65] for the modeling, and (ii) the STRIPS planning approach for the automatic service composition. Essentially, the approach takes as input a feature model configuration, satisfying structural and integrity constraints of a feature model, as defined by the user. Indeed, feature models are strategically used because of their high level of understandability by the users, since they allow them to specify their requirements by defining the variability of a product family. Then, to obtain executable service mashups as output, the approach automatically generates a service composition expressed in WS-BPEL, which can be readily executed by exploiting existing WS-BPEL engines. In particular, the proposed approach first creates a workflow model that consists of all the features present in the domain model (i.e., the feature model configuration and the connected services) through an AI planning problem. Moreover, the defined workflow is further optimized, through the introduction of parallelism, before being finally converted into WS-BPEL.

A responsive decentralized composition of service mashups for the Internet of Things is proposed in [93]. In fact, in last years, to facilitate the development of IoT application, different mashups editors have been proposed, such as, for instance [94]. These are domain-independent tools providing developers a visual support abstracting both devices and services

they can compose together. Other approaches that are taking place, are those services enabling end-users to create by themselves IoT mashups, by using event-conditions-actions (ECA) rules. Among these services, IFTTT [19] is probably the most famous one. In [93], the authors objective is that of overcoming the static nature of IoT applications that, although highly responsive, are usually based on pre-compiled mashups, being thus inflexible. As in our approach, the challenge that the authors want to address relates with the highly dynamicity and openness of IoT environments, made by a large number of devices that may become available or unavailable at run-time. The idea on which this work relies on is very simple and powerful at the same time. It is based on a decentralized goal-driven composition of pre-compiled service mashups. More in detail, to provide an abstraction of the environment, on top of which applications can be defined, the approach uses socio-technical networks (STN). A STN is a dynamic system of people and things interrelated in a meaningful manner via specific relations. Then, both people and things can enter or leave the STN, modify their relations or interact with each other via messages. In particular, people and things are modeled as goal-driven software agents and they are further equipped with plans that they exploit to cooperate with other agents, to dynamically compose IoT mashups and, thus, achieve their goals.

In summary, compared to our approach,[17] underlines the importance of the context-awareness and adaptivity requirements when defining service mashups that otherwise tend to be unsuccessful or misaligned with their execution environments. The underlying idea of the work in [92], instead, is quite similar to the idea behind our approach. Indeed, also in this work we can see the combination between a sound modeling approach, based on SPL feature models, and a planning approach, in order to exploit the potentialities of both. Moreover, both [92] and [93] are based, like our approach, on the abstraction of the referring environment allowing to abstract composition requirements. In particular, this is done with feature models, in [92] and with STN in [93]. However, in the majority of service mashups approaches, the trade-off is that the corresponding applications rely on static mashups that cannot adapt to services entering and leaving

[19] `https://ifttt.com/`

the environment at run-time.

Lastly, besides short-term adaptation, *long-term adaptation* is also part of the adaptation, meant as an umbrella term. In service-oriented computing, long-term adaptation is mostly known as **service evolution**. Service evolution can be seen as a special case of software evolution, which is known to be an inevitable and critical part of the software life cycle, as explained by Lehman in his eight laws of software evolution formulated in [95]. Researchers have spent significant effort on investigating methods and techniques for the management of software evolution. In particular, research into the evolution of component-based systems, such as [96], can be seen as a precursor for research on service evolution. However, traditional component-based evolution works under a "closed–world" assumption, which tends to limit evolution to a few entities, and in a centralized manner. This is not feasible for service evolution scenarios since service–oriented computing happens in an "open world" [5], and calls for the distributed, coordinated evolution (*co-evolution*) of multi-party applications.

Service evolution has been studied by several researchers. Papazoglou et al. distinguish between *shallow changes* and *deep changes* [97],[98]. The problem of shallow changes in service evolution, i.e. small-scale, incremental changes that are localized to a service and/or are restricted to the consumers of that service [98] is considered solved. However, none of the existing approaches has solved the problem of deep changes, i.e. the multi-step co-evolution in a graph of interdependent services. In [97], the author has classified evolutionary change types in terms of their scope and effects, and pointed out how they pose different challenges and impact in very different ways the maintenance of service–oriented applications.

There are several approaches that support the maintenance and evolution of service–oriented software. For instance, Ryu et al. [99] describes an approach to handle long-running clients instances when the business protocol of the service provider needs to change. This is an evolution problem relevant to our work, in view of an extension of our solution to deal with service co-evolution (already started in [34]), since we also consider service protocols, which are a first–class element of the Domain Objects specification. In [100], the authors similarly propose an approach that allows service providers to break their backwards compatibility for evolution pur-

poses, by managing the conflicts with the service consumers using a set of refactorings addressing complex changes. *WSDLDiff* [101], instead, analyzes the types and the frequency of changes affecting WSDL interfaces, and provides recommendations to service consumers; its aim is to prevent reliance by those consumers that are particularly unstable because they often undergo evolutive maintenance.

The works above offer support to different facets of the evolution problem. However, they do not directly address deep changes [98], and co-evolution across that chain. This remains largely an open problem, which is of increasing importance because of the ever–growing complexity of contemporary service landscapes. *WSDarwin* [102] is an Eclipse plug-in that attacks the problem of deep evolutionary changes, since it automates the adaptation of the service clients across the co-evolution chain, when structural or behavioral changes are made, to the service interface specifications. However, WSDarwin is a maintenance support tools that operates exclusively at development time.

Our preliminary work on a distributed service co-evolution approach [34], based on an extension of the work discussed in this dissertation, emphasizes the decentralized, dynamic and collaborative management of service co-evolution; it focuses on alleviating the run-time impact of deep evolutionary changes, by using automation and operating on-the-fly whenever the semantics of the changes permits. Our solution focuses on structural and behavioral changes, and do not address at this stage changes deriving from business policies or regulatory compliance issues.

The level of evolution automation we obtain with our proposal in [34] is similar to what is advocated by research in self–adaptive software systems, in particular compositional (a.k.a. architectural) adaptation approaches, such as RAINBOW [103] and MUSIC [104]. MUSIC, in fact, inspired the design of our service co-evolution framework, since it already supports service-based adaptation [105], although in a context of anticipated adaptation, with strategies that aim at improving the utility of the system, in response to a repertoire of known situations.

In conclusion, the need for advancements in service evolution support is undisputed, just as for all software. A number of open questions remain, including issues of scalability, transactional update guarantees, security,

and delineate a rich research landscape.

2.4 Discussion

In conclusion, huge work has been done in the literature in the research areas of modeling and adaptation of service-based systems and applications, as surveyed in this chapter. Although the existence of all these works, the main limitation that we can observe is essentially represented by the fact that each of them deals with one or a few specific aspects in the considered research areas (e.g., improving reuse, decreasing development costs, increasing the system's flexibility, system's evolution, service composition, etc).

To the best of our knowledge, it does not exist a *comprehensive and coherent approach* defining a *complete life-cycle* for the continuous development and deployment of *adaptive* service-based systems with a huge degree of *flexibility* and *extensibility*. An approach allowing the continuous integration of newly available services and the operation of systems in *open* and *dynamic* environments.

We recall that we concentrate on systems and applications that rely on a dynamic set of autonomous and heterogeneous services that are composed dynamically without any a-priori knowledge between the applications and the exploited services. This strongly supports the loose coupling of services and allows systems to effectively execute under dynamic circumstances.

These premises highlight the need for approaches supporting both the *definition* and the *application* of systems adaptation, from the early stages of systems design.

To sum up, the solutions in the literature are limited for one or a combination of the following crucial aspects.

- *Low applicability in open environments.*
- *Portability.* What is expected from modern systems is that they are deployable in different environments without an ad-hoc reconfiguration from the developers.

- *Context-awareness.* This limitation affects especially those approaches meant for the application in closed environments. The deriving systems are not able to take into account the current state of the environment that, instead, is strongly subjected to changes.
- *Distance from real services and real execution contexts.* This limitation is often observable in research approaches that, as a consequence, struggle to get out of the realm of research. To the contrary, industrial approaches often suffer from the opposite problem, since it can be that they strictly relate to services of the companies for which they have been defined.
- *Detachment of adaptation requirements from external services specification.* This limitation is also connected with the service loose coupling principles and it causes systems to be too much rigid.
- *User centricity and personalization.* Most approaches focus on business-centric systems, while modern environments are asking for a shift towards user-centric systems.
- *Support for continuous development.* This limitation arises as a consequence of the openness of the environments. The continuous entrance of new services requires for their continuous development and deployment.
- *Scalability of the adaptation techniques.* Considering the huge and increasing amount of services, the techniques supporting the adaptation of systems must be scalable to face the environment dimensions.

This motivates our work that aims at defining a *novel design for adaptation approach of service-based systems.* The approach has been inspired from [24] suggesting that mechanisms enabling adaptation should be introduced in the life-cycle of systems, both in the *design* and in the *run-time* phases. Indeed, our proposal aims to be the implementation of the idea that in modern service-based environments adaptation cannot be considered an exception to be handled, but instead, systems must be adaptive by design.

Chapter 3

Design for Adaptation of Service-Based Systems

The goal of this chapter is that of introducing the reader to our approach for the development and execution of adaptive by design service-based systems in the IoS. In particular we present the *overall life-cycle* of the design for adaptation approach of systems operating in open and dynamic environments. The subsequent chapters, instead, are devoted to present in more details the new design approach for modeling adaptive by design service-based applications, in chapter 4, and how the defined applications operate at run-time, in chapter 5.

In the reminder of this chapter we (i) introduce a motivating example, in Section 3.1, that will guide the explanation in the subsequent chapters; (ii) we illustrate the overall life-cycle for the development and execution of service-based systems in the IoS, with the focus on its key characteristics, in Section 3.2.

3.1 Motivating Example: Travel Assistant Scenario

The scenario that will drive us throughout this dissertation comes from the mobility domain. In addition to being a particularly suitable domain to show the challenges of dynamic environments, the mobility domain is also very relevant both at European and world level. For instance, many past and running projects have been funded with the aim of providing solutions to support and stimulate a more integrated and sustainable mobility

(e.g., [6, 7, 8]). The scenario is concerned with the management and operation of mobility services, within a smart city as well as among different cities/countries. Nowadays, a huge number of users is constantly on the move. They dispose of a large offer of mobility services. These services may differ depending on diverse aspects, such as the offered functionalities, the targeted users, the provider, the geographical applicability scope, etc. In addition, mobility services span from *journey planners* for finding traveling solutions between two or more given locations, to *specific mobility services*, such as those referring to specific transport modes (e.g., bus, train, bikes) or provided by specific transport companies. Moreover, an emerging trend is that of *shared mobility services* that are based on the shared use of vehicles, bicycles, or other means. All these services can involve or refer both to *public* and *private* transportation services. Furthermore, mobility services can offer disparate functionalities (e.g., journey planning, booking, online ticket payment, seat reservation, check-in and check-out, feedback evaluation, user profiling, and so on). Some functionalities may be peculiar to specific services and/or require particular devices (i.e., the need for unlocking a bike from a rack is peculiar for bike-sharing services, and a smart-card might be needed to do it). Often, users must interact with different applications to exploit the different functionalities required to accomplish a journey (e.g., a journey planner for planning, one or more mobility services for booking). In addition, these services are made available through a large variety of technologies (e.g., web pages, mobile applications), with different constraints on their availability (e.g., free vs. pay).

From a high level point of view, a journey organization, as seen from a user perspective, consists of a set of different mandatory and/or optional phases that must be carried out (e.g., planning, booking, check-in, check-out). In Figure 3.1, for instance, we give a high level overview of the possible phases that might be part of a journey organization, together with some of the possible activities that can be run in each phase.

While these phases define *what* should be done, *how* they can be accomplished strongly depends on the users requirements and preferences, and from the specific procedures that need to be followed, as provided by the available mobility services that will be effectively involved in the journey. Firstly, a user plans her journey looking for the available (multi-modal)

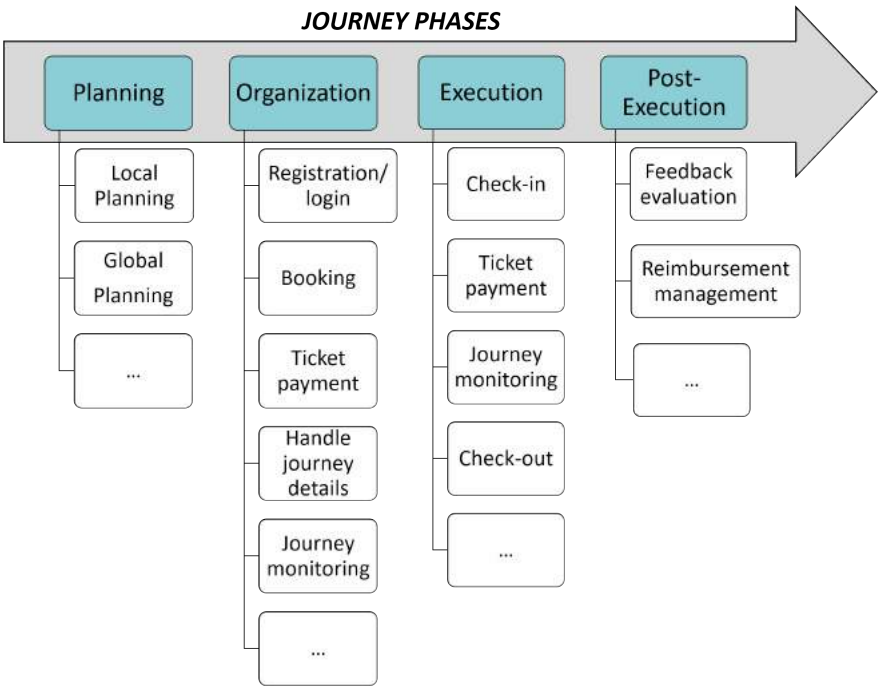


Figure 3.1: Overview on the different phases of a journey organization.

alternatives satisfying her needs. Depending on her departure and destination locations, a journey can be *local*, in the context of a city, or *global* involving different cities/countries. Then, based on the user's preferred alternative, this can be a multi-modal solution involving different transportation means, each requiring for different procedures to be followed (e.g., a bus journey differs from a air journey). Moreover, procedures can differ also among services of the same kind (e.g., the local bus offer in a city generally does not require a booking, while long distance bus services do). Once a planned journey is going to start, the overall procedure continues through the journey execution. Again, the user must behave according to the execution rules relating to the selected alternative and the involved services (e.g., for an air journey a check-in at the departure gate is required). In the case in which extraordinary events affect the journey by altering its normal execution, it can be re-planned and recovery solutions can be suggested to the user. Additionally, even when a journey is over, mobility services can provide further functionalities (e.g., request of feedback on the offered service, offer of promotions for future journeys, management of reimbursement requests from users).

In this context, our goal is to implement a system, such as a *Travel Assistant*, able to support users in the overall organization and execution of their journeys, for the whole travel duration. To this aim, different mobility services need to synergistically cooperate, while adhering to their own procedures. While the idea of an intelligent travel assistant has already been figured out in the past, as for instance in [106], our opinion is that we are still far from making it happen. In particular, the system needs to satisfy different *requirements*, as listed in the following:

heterogeneity awareness The system must consider the heterogeneity of the services in terms of technologies, or in terms of offered functionalities (e.g., REST API vs. SOAP, online booking vs. on board ticketing);

autonomy awareness The system must take into account the autonomous nature of the services involved;

openness awareness The system must be capable to operate in open environments with continuously entering and leaving services, which are not known a priori (e.g., a new ride-sharing service is available in the city);

interoperability The system must be capable to propose complex solutions taking advantages of the variety of services (e.g., a user needs of a unique solution with her booking, payment, train journey, and taxi ride);

customizability The system provide users with personalized solutions (e.g., a wheelchair user has preferences on transportation means and stops);

information accuracy The system must provide up to date and reliable information and solutions (e.g., temporary changes on a bus route);

context awareness The system must take into account the state of the environment (e.g., strikes, bad weather, roadworks).

adaptivity The system must be able to react and adapt to changes in the environment that might occur and affect its operations (e.g., a strike

affects the user's train journey and the system offers a new journey plan).

portability The system must be deployable in different environments without an ad-hoc reconfiguration from the developers (e.g., the travel assistant must be usable in Trento as well as in Paris).

The idea behind the travel assistant is that of providing a solution for enhancing mobility services interoperability through their run-time and context-aware discovery and composition, in order to exploit their potentialities and overcoming their diversity. This would allow users to get personalized solutions for their journeys, by interacting whit a multitude of mobility services while using just one application and in a complete transparent way.

Discussion

One of the motivations that drove this work is that of solving some of the complex real-world challenges, by trying to overcome the current criticality related to the service-oriented computing paradigm, arising especially when it is applied to real-world scenarios. Despite the innovations and advances of service-oriented computing, the evolving environments in which service-based systems live introduce new requirements that still need to be achieved, as it emerged from the scenario. In particular, to guarantee both the *portability* of systems and their *context-aware applicability* and operation in open and continuous evolving environments, the requirement of separation of concerns must be accomplished. For instance, the *application logic* must be kept separate from the *adaptation logic*. While the former can be predefined, the latter is strongly affected by the run-time operational context of the system, thus it cannot be foreseen when systems are designed. Moreover, without the opportunity to know a priori the involved services and the users' needs, the detachment of the adaptation requirements from the external services specification is crucial.

The identified requirements typically ask for the definition of several levels of abstraction, when modeling complex systems. This helps not only to address separate system requirements and concerns, but also to deal with and integrate different knowledge over the whole system.

As a consequence, the intuition behind our approach mainly consists in the idea of thinking separately about *what* a system must do and *how* it does it. For instance, considering the travel assistant, its provided functionalities (e.g., journey planning, online ticketing, journey monitoring, and so on) represent *what* the travel assistant does. One or a combination of these functionalities might help the users to reach their *goals* such as moving around a city, or different cities, getting information about transportation means, their timetables and their costs, recovering from unexpected changes in their journeys, and so on. Contrariwise, the concrete implementation of these functionalities, representing *how* they might be effectively accomplished, can vary depending on the execution context, since it can be provided by disparate real-world services, or a composition of them, in a completely dynamic way. For instance, let us consider a trivial but exhaustive example, in a city context. A bus journey whose execution allows the user to move from a source to a destination point, by reaching her goal, will be concretized according to different bus companies services in different cities. Indeed, while the goal of moving around in a city is quite general, each city has its own local transport service, involving diverse transportation companies, that can offer various mobility solutions. And if we move from a *local* (e.g., a city) to a *global* (e.g., the world) scope, the scenario becomes actually more complex, due to the increase of possible alternatives and existing services.

3.2 Lifecycle of the Approach

In this section, we present the life-cycle of our approach for modeling and executing adaptive by design service-based systems that are able to meet the requirements highlighted in the motivating example. The life-cycle, graphically depicted in Figure 3.2, gives a complete overview of the different perspectives of the approach (i.e., system models, adaptation, interaction), the involved actors (i.e., platform provider, service providers, end-users) and an abstraction of the main needed artifacts. While illustrating it, we start to introduce the key elements required for the modeling and operation of systems, such as the travel assistant. In Figure 3.3, instead, we further

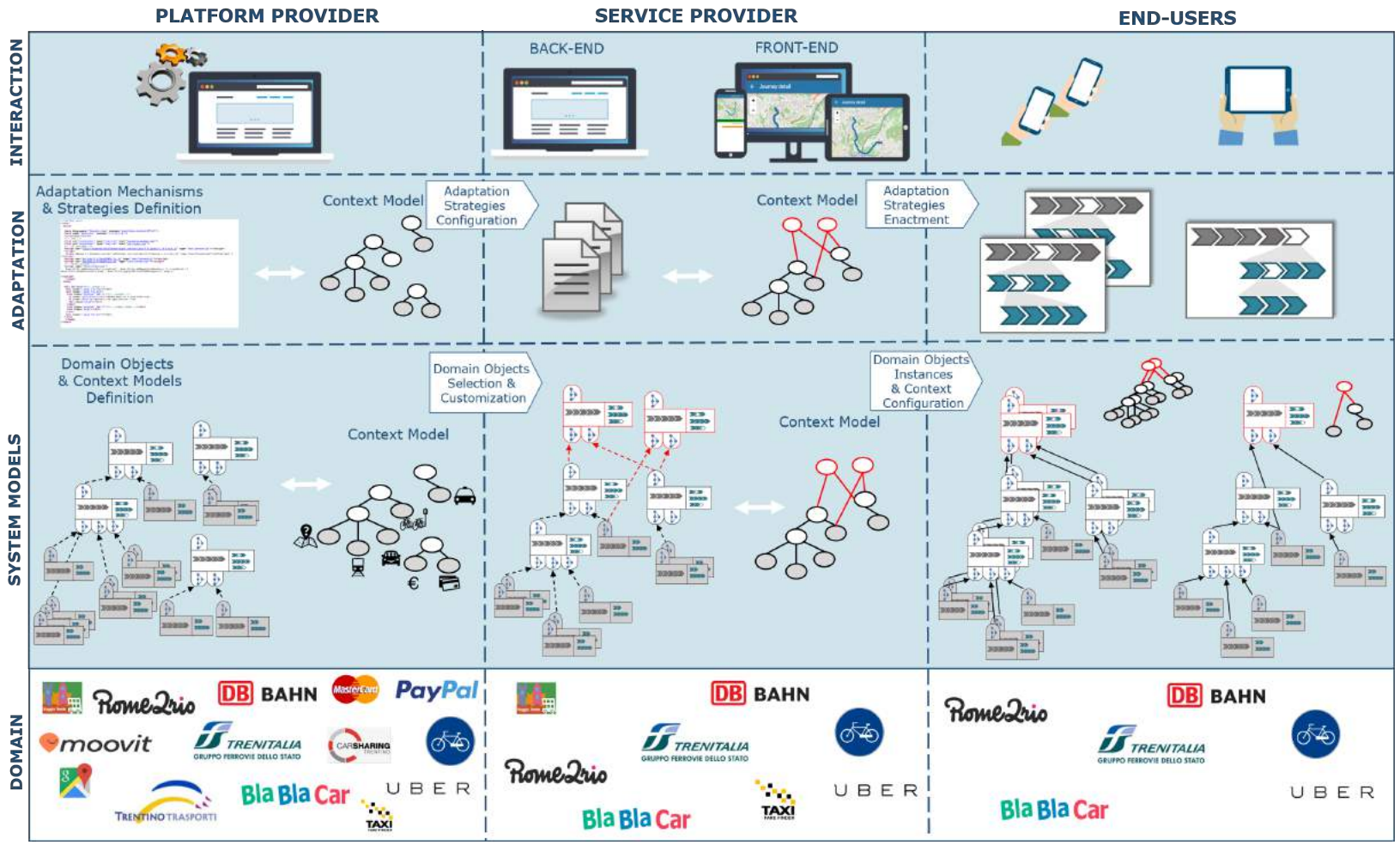


Figure 3.2: Life-cycle of the design for adaptation approach.

specify the activities that are performed by the different actors regarding the definition and exploitation of the *system models* and the configuration of the system *adaptation*. We give a high level description on how these activities are executed, their input and output artifacts and the connections among them. Details will be given in the following sections.

We remark here that the approach is domain-independent and it can be applied in multiple domains (e.g., logistic, traveling, entertainment). To simplify the approach life-cycle specification, from now on we refer to the *mobility domain*, which also represents the domain of the motivating example of this dissertation.

The presented approach sees the involvement of three main actors. The *Platform Provider*, with his team, is in charge to realize, maintain and provide to third parties a comprehensive platform allowing them to build and execute adaptive service-based applications on top of it. This can be done by capitalizing on the services and the enablers exposed by the platform, for both the design and operation of their applications. Indeed, on top of the platform, different *Service Providers* can realize their own applications (or simply value-added services) as the dynamic combination, the extension and the customization of the services the platform furnishes (e.g., the services available in the environment). These value-added services can become part of the platform as well, making it continuously evolving. Service providers will also rely on the platform for the runtime operation of their applications, by exploiting advanced execution and adaptation techniques as provided by the platform itself. Finally, the *End-Users* are those that effectively make use of the different applications in their daily lives, by interacting with their interfaces (e.g., web pages, mobile apps, chat-bots).

In the following, each subsection is devoted to a particular perspective of the overall life-cycle (e.g., a row in Figure 3.2). For each perspective, we also highlight the view of the different involved actors (e.g., a column in Figure 3.2).

3.2.1 The System Models Perspective

From a system's modeling perspective, each actor has a different view on the system's models and is differently involved in the system's development

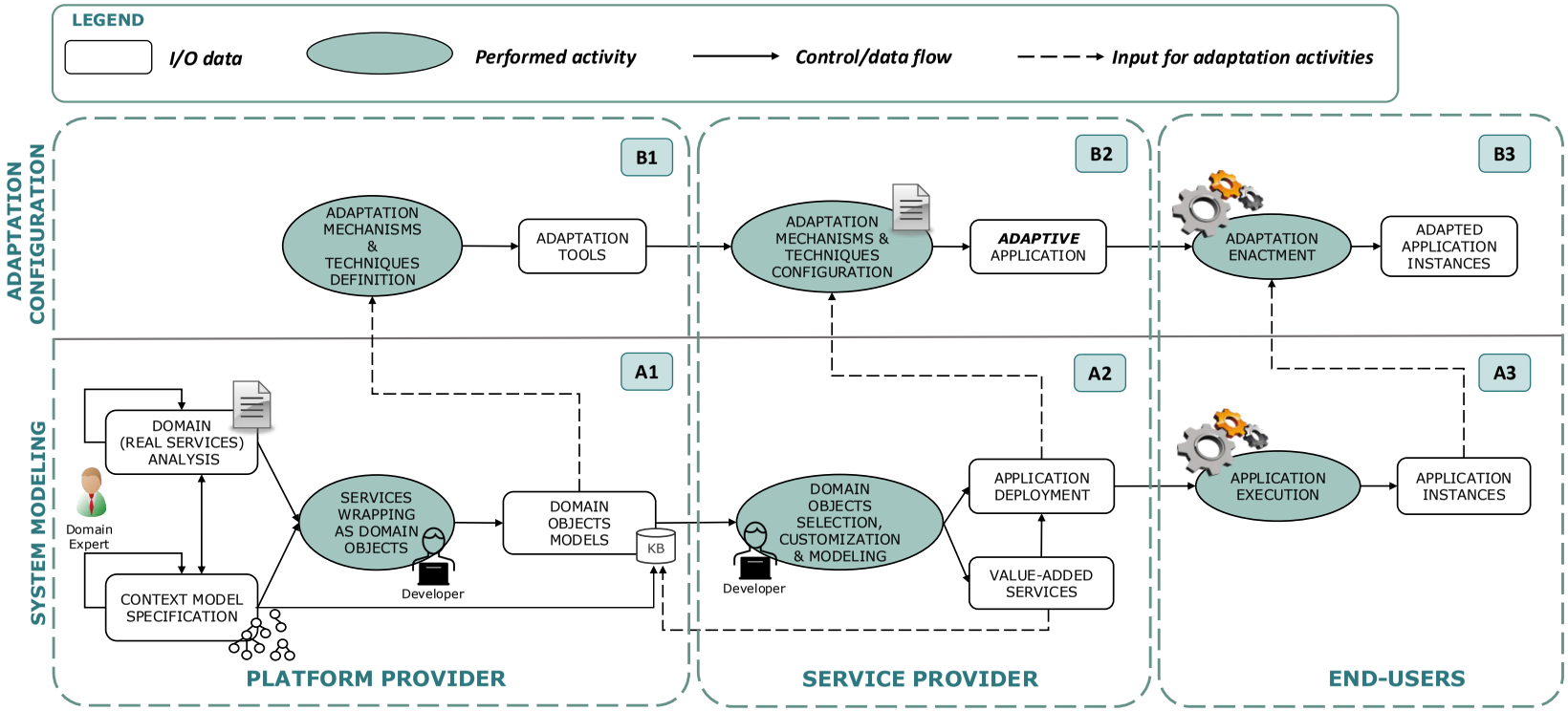


Figure 3.3: High level view of the system's modeling and adaptation configuration processes.

and operation. We start by presenting the view of the platform provider.

Platform provider view. In the following, we are going to specify the required system's models, what they represent and how they are defined, with the aim of realizing adaptive by design service-based systems. We say that the design of the system (or the basics for a system) is a contribution of the platform provider and it is made by two distinct but correlated models: the *Domain model* and the *Domain Objects model*.

Given the specific domain in which the platform provider wants to instantiate the system under development (e.g., mobility), the **domain model** is specified by domain experts and it describes the operational environment of the system. In particular, the domain is modeled as a set of *domain properties* describing specific concepts of the domain (e.g., bus journey, ticket booking, journey monitoring, journey planning).

Domain experts specify both *domain-specific* concepts, such as journey planning, train journey, bike sharing journey (in grey in Figure 3.2 – on the bottom left side), and *general-purpose* concepts that are implicitly part of the domain and that can also be defined on top of the domain-specific ones, such as the user profile management, the user tracking and so on (in white in Figure 3.2, on the bottom left side – system models / platform provider).

Domain properties are modeled as *State Transition Systems* (STS) evolving as an effect of the normal execution of service-based applications, or because of exogenous events in the operational context [107], [108].

We illustrate these notions with a simple example.

Example. *A simple but representantive example of a STS is shown in Figure 3.4. It refers to the abstract definition of the* ride sharing *concept, with its possible states and transitions defining the events allowing the domain property to evolve from a state to another.*

Furthermore, another important contribution from domain experts is an accurate analysis of the available services that are part of the specific

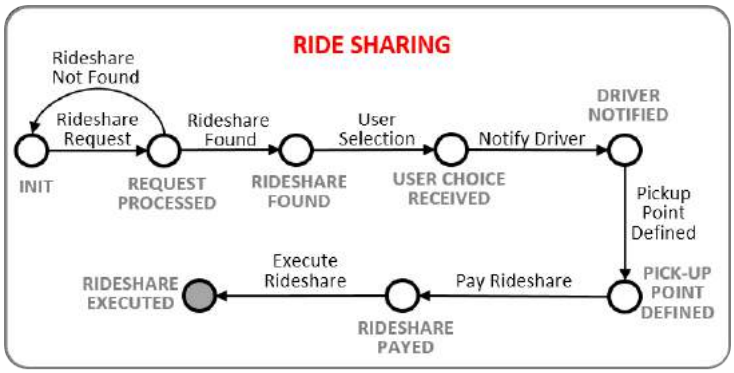


Figure 3.4: Ride sharing concept as state transition system.

domain. For this analysis, different aspects can be taken into account, such as the technologies with which services are implemented and exposed to the outside (e.g., REST API, SOAP), constraints on their availability (e.g., free vs. pay), potential dependencies among them (e.g., to unlock a bike of a bike-sharing service a smart-card can be required) and their specific functioning. As outcome of this analysis, the domain experts release a high level description of the features, behavior, usage and offered functionalities of the analyzed services.

The idea of keeping separate the domain model from the concrete implementation of services, besides responding to the need of abstracting from the real services, is also inspired by the domain-driven design methodology [109]. Indeed, the domain-driven design approach is particularly helpful when designing systems relying on complex domain specific knowledge, as service-based systems often are. Behind the principles it promotes, there is the idea of leveraging on the skills of the designers, developers and the domain experts in order to create a scalable and accurate solution for a domain specific problem. This must be done also by facilitating the communication among the involved participants. To this aim, the domain-driven design approach devises the use of a common language and a single model that reflects a shared understanding of the domain under analysis across the different experts. We believe that the domain model we just introduced acts as the shared model on which designers, developers and domain experts agree and that guarantees as much as possible a common understanding of the domain in which the service-based system under de-

velopment will live. Moreover, having a single domain model allows for the reduction of chances of errors, since the design of the domain objects model, which we introduce in the following, will be a consequence of the carefully designed domain model.

At this point, both the domain model and the description of the services analysis constitute the input for the activity of defining the *domain objects model*, accomplished by the system's developers (Figure 3.3–label A1).

A **domain objects model** represents a uniform way for defining autonomous and heterogeneous services as domain objects, each implementing a specific concept in the domain model. In other words, developers wrap-up the services identified by domain experts as the *concrete implementations* of the *abstract concepts* in the domain model.

In particular, *domain-specific* concepts are usually implemented by domain objects wrapping back-end services requiring for the interaction with third-party systems and devices (e.g., train mobility services). To the contrary, *general-purpose* concepts (e.g., user profiling) might be realized by defining new domain objects, by also exploiting the functionalities provided by the domain-specific ones, if needed. To better clarify the connection between the domain and the domain objects models, we give some example in the mobility domain.

Example. *If we consider a domain-specific concept in the domain model, such as the* train journey*, many train services provided by different train companies might represent diverse concrete implementations of the same abstract concept. Indeed, even furnishing the same service, they might do it by following completely diverse procedures and exploiting different technologies, thus actually providing diverse implementations. For instance, some of them can ask for a user login, preceded by the registration of the user to the service, while others can be used without a user's identification. Eventually, all these services can be wrapped-up as domain objects, each providing its own implementation of the same train journey concept and exposing their functionalities.*

Therefore, the wrapping of heterogeneous services as domain objects implementing the abstract concepts in the domain model allows us also

to overcome the typical mismatch among services' interfaces. In other words, especially if we focus on domain objects wrapping diverse services but implementing the same domain property (e.g., two different ride-share services implementing the same ride sharing property in Figure 3.4), we can notice that, while the domain property abstracts away from any specific details about services interface, process or data, each domain object implementing it will then provide its own implementation of the domain property, by reflecting the real service it wraps-up. In this way, these diverse services can be used indifferently one in place of the other without giving rise to semantic data interoperability problems.

As noted, the domain object concept is central in our approach. The core idea is to factorize the capabilities offered by service providers as a set of building blocks (i.e., domain-objects), which can be easily combined with each other to give rise to composite services. Relevant (composite) services will be published so that stakeholders can personalize and turn them into new available services (i.e. applications). Thanks to this general approach, we can facilitate services *integration* and *interoperability*.

In particular, the domain objects model allows developers to define services and applications by specifying (1) their offered functionalities (so-called **fragments** [110]) defining *what* a service does and, (2) their application logic (so-called **core process**) determining *how* a service effectively realizes its functionalities.

Now, the strength of the model allowing services to be adaptive by design relies on the way in which fragments and core processes (from now on, simply processes) are modeled.

Indeed, our approach allows the definition of **dynamic behaviors**. In other words, the developers can define **abstract procedures** saying what might be done, but without specifying how to do it, thus leaving the context-aware specification of these procedures to the runtime execution of the system. Dynamic behaviors are defined by using the **abstract activity** construct, that is an activity specified only in terms of an abstract goal representing a desired domain configuration to be reached. Its concrete implementation will be shaped at runtime by se-

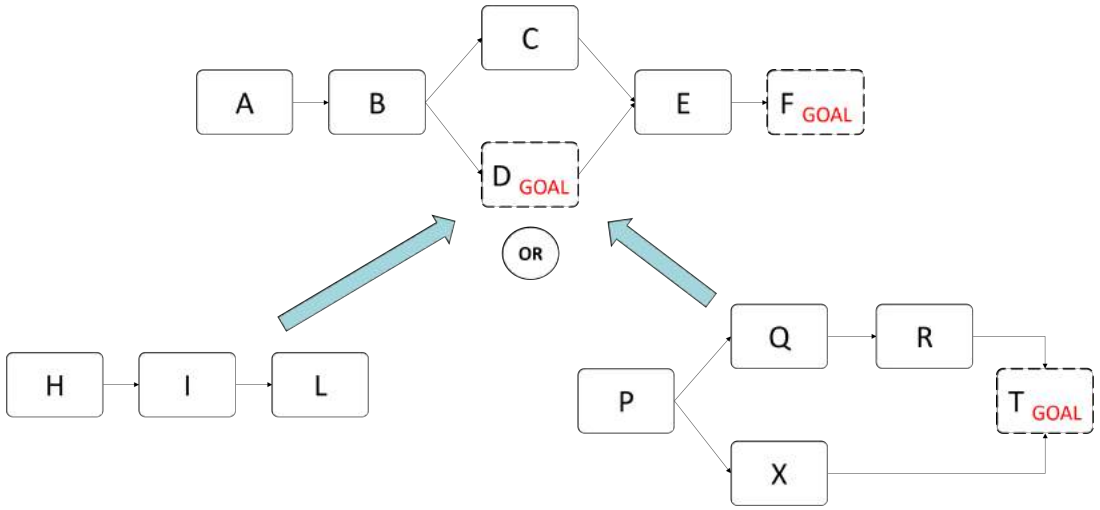


Figure 3.5: Example of a customizable dynamic behavior.

lecting fragments offered by other domain objects in the system, whose execution lead to reach the goal.

A graphical example of an abstract activity that might be specialized in different ways, that is by exploiting diverse functionalities, is shaped in Figure 3.5. This specialization approach represents a *flexible connection strategy* among domain objects that enhances services interoperability and provides a lightweight mode to define relations among services. To give to the reader a simple example, let consider our travel assistant.

Example. *Among its functionalities, it exposes a fragment allowing users to plan their journeys from a source to a destination point. It does not implement the journey planning service by itself, but it relies on the available journey planners existing in the mobility domain environment, that have been previously wrapped as domain objects. These journey planners, besides furnishing quite the same service, can have requirements on their utilization (e.g., global vs. local planners). As a consequence, the travel assistant can specify an abstract* plan journey *activity. Then, during its execution, depending on the specific needs of the user, his/her location and the journey planners effectively available and useful in that domain, the appropriate fragments will be selected and injected in place of the abstract activity, to provide a personalized and accurate solution to the user.*

We want to highlight here that, differently from traditional service in-

terfaces, fragments allow for a partial specification of service interfaces (because of their use of abstract activities), enabling to model *adaptable processes*, that is processes suitable for the execution in dynamic environments. Moreover, the use of fragments and abstract activities represent a way to enable the adaptation from the design phase, which is one of the requirements argued in [24], where the authors suggest the requirements needed for realizing modern service-based adaptive systems.

Eventually, all the domain objects models defined during the system's modeling phase contribute to enrich the *system knowledge base* where they are stored.

The whole system is finally represented as a **dynamic network of domain objects**, interconnected via their offered and required functionalities, on the need.

We give a simple example to illustrate this feature.

Example. *For instance, in the mobility domain, a system such as the travel assistant, will expose services for planning a journey, verify and booking multi-modal solutions, monitoring whatever changes and so on, by relying on the various domain objects available in the system's knowledge base and offering the functionalities the travel assistant needs. Different execution of the travel assistant will give place to different network of domain objects.*

We want to highlight here that the activity of wrapping services as domain objects (or defining new ones) is not executed only once, and certainly not only during the initial design of a system. To the contrary, it is a continuous running activity, due to the continuous discovery and availability of new services. Moreover, this activity can be performed as a collective co-development process [111, 112], in a crowd-sourcing style [113, 114], where each developer contributes to add new interesting services, thus enriching the system knowledge base.

For these reasons, we say that our approach supports the **continuous development** of service-based adaptive systems.

In the software engineering field, we are used to see the expression *continuous development* referring to the realization of modern software systems.

Although it is used in many different contexts and with different meanings, it essentially refers to the increase of automation of the overall software development cycle, in some or all its phases. In this dissertation, we consider the continuous development expression as an umbrella term comprising several phases of the iterative software application development including continuous integration [115], continuous delivery [116] and continuous deployment [117]. The continuous development process notably sustains system providers and developers in managing the systems maintenance and development activities that are particularly challenging in open environments. Our approach particularly supports the definition and deployment of new domain objects allowing the system to evolve.

To sum up, we have given an overview on the lightweight and flexible model for specifying adaptive by design service-based systems based on the domain object concept. In the next paragraph we will see how service providers can exploit representative (composite) services published as domain objects through the platform and customize them.

Service providers view. The role played by the service providers can better help the readers in understanding the strengths of the approach. Their role is that of using and exploiting the tools, the engines and the models provided by the platform, in order to define, develop and execute their own service-based systems on top of it. This can be done by *selecting* and *customizing* the already available domain objects and by *defining new value-added services as domain objects* (in red in Figure 3.2 at the system models level under the service provider column), together with the corresponding new domain concepts they implement, if not existing in the domain model.

Example. *For instance, in the mobility domain, the platform will probably expose domain objects referring to journey planning services, different transport means services, weather forecast services, online payment services and so on. A service provider can select and customize all or a subset of these services and exploit them to define a new service, such as a travel assistant, implementing a new concept, also defined as a combination of existing ones. For instance, the new travel assistant service will implement the* travel assistance *concept, that might rely on the* planning, booking *and*

travel execution *concepts in the domain model.*

As stated before, also the domain objects defined by service providers can be stored in the system knowledge base and made available to the outside. In this way they contribute to the continuous development of adaptive by design services and corresponding systems (see Figure 3.3 – label A2). Moreover, service providers can decide to develop and release their newly defined services, by using whatever technologies independently from the platform, such as a chat-bots, web-pages, mobile apps, web applications, etc (see Figure 3.3 – label A2). For instance, the travel assistant could be released as a mobile application to better following users as they move around.

End-users view. End-users are the final beneficiaries of the deployed service-based applications. Different application instances will be instantiated for different users and each instance will be characterized by its own *network of domain objects instances* and its *run-time domain configuration*, directly affected by the user it relates to, his/her needs and requirements. The domain objects network is made by instances of the domain objects corresponding to the services effectively exploited by the user. A domain configuration, instead, is a snapshot of the domain at a specific time, illustrating the current status of all its domain properties. Indeed, domain properties evolve as an effect of the execution of activities in fragments (e.g., the journey status can change to *planned* as an effect of the execution of a *journey planning* fragment), during the normal execution of the system, or as a result of exogenous context changes.

3.2.2 The Adaptation Perspective

In this section we describe the adaptation perspective in the life-cycle of our approach. In order to make service-based systems *adaptive-by-design*, two conditions are required: (1) the models adopted for the systems design must facilitate the definition of dynamically customizable systems behaviors, through the adoption of adequate constructs; (2) the approach must implement or exploit adaptation mechanisms and strategies whose application allows for a context-aware and dynamic systems adaptation, during their execution.

Platform provider view. From the adaptation perspective, the platform provider must supply all the tools and enablers allowing both the platform users (i.e., service providers) to define adaptive applications on top of it, and the applications to effectively perform the adaptation, when executed. In other words, the *adaptation mechanisms and strategies* used by the platform must be exposed in such a way that external users can benefit from and exploit them.

In the following, we are going to see the specific constructs used during the system's design, allowing the system adaptation to be performed, through specific adaptation techniques. We want to highlight here that keeping separate the *domain model* from the *domain objects model* allows our approach (i) to implement the separation of concerns requirement (i.e., adaptation vs. application logic), and (ii) to simplify the specification of processes (i.e., fragments and core processes) that must operate in a dynamic environment. Indeed, the adaptation configuration of systems strictly relates with the domain model and its connection with the domain objects model.

Being an abstract representation of the operational environment, on top of the *domain model* it is possible to define *rich* and **abstract composition requirements**. In fact, the framework allows for expressing realistic and complex composition problems as rich control-flow requirements [118, 119]. At the same time, these requirements are abstracted away, that is, detached from services implementation. This allows for extracting them automatically, as well as for reusing the same requirement for different services or in situations in which the implementation of services changes constantly.

At a system modeling level, the definition of composition requirements, used to configure the system adaptation, is realized by defining annotations labeling fragments and core processes defined in the domain objects making the system. Indeed, the language used to model the adaptable processes, besides allowing for the definition of classic work-flow language constructs (i.e., input, output, concrete, user activities, control-flow constructs), gives also the possibility to link the domain objects processes with the system's domain model, by annotating the processes activities. These *annotations*

can be of different types.

Goals describe the desired domain configurations relating to abstract activities. Other relevant annotations are **Preconditions** and **Effects**. Briefly, preconditions are used to constraint the execution of the activity it annotates to a specific domain configuration. Effects, instead, model the expected impact of an activity execution on the system domain, and are used to perform automatic reasoning.

Examples of annotation are given in the following illustration.

Example. *For instance, the goal defining an abstract activity such as* Book a Rideshare *will be defined over the* Ride Sharing *domain property (in Figure 3.4) as follows:* `Goal: Ride Sharing = PICK-UP POINT DEFINED.` *Then, if we think to a fragment provided by a ride-sharing service and allowing to book a rideshare, a precondition annotating its first activity might be as follows:* `Prec: Ride Sharing = USER CHOICE RECEIVED`*, stating that the ride-sharing mobility service must be in the right configuration to accomplish the booking, that is, the user must have chosen a specific solution to be booked. If violated, it means that the system is not behaving as expected and thus run-time adaptation is triggered. Eventually, in the* Ride Sharing *domain property effects are those labeling the transitions among states. The effect of executing the before-mentioned rideshare booking fragment might be* `Eff: Ride Sharing.pickupPointDefined.`

One of the strengths of our approach, relies exactly on this way of modeling processes in each domain object. It allows for simplifying the specification of *adaptable processes*, by releasing the developers from the activity of thinking about all the possible alternatives that might occur, with respect to changes in the domain, availability of fragments and so on. Moreover, besides being a time-consuming and error-prone activity, in most cases it is not possible at design time to handle adaptation needs and extraordinary situations.

For the resolution of adaptation needs occurring at run-time, our ap-

proach exploits advanced **techniques for dynamic and incremental service composition** [25, 120] based on AI planning [47].

In particular, these techniques also deal with services represented as processes, thus resulting particularly suitable to be used in conjunction with our domain objects model and its constructs. The exploited approach implements different *adaptation mechanisms and strategies* that can be used to handle the dynamicity of context-aware adaptive systems. Without going into details, and referring to the chapter 5 to see these mechanisms in action, we say that the adaptation framework that we exploit [25, 120] deals with three kinds of adaptation needs:

- The first one refers to the need for refining an abstract activity when executing a process. This is made by triggering the ***refinement mechanism*** whose execution allows the framework to automatically find and compose available fragments provided by other domain objects in the system by defining an executable process whose execution guarantees to reach the goal of the abstract activity.
- The second refers to the violation of the precondition of an activity. In this case the ***local adaptation mechanism*** is performed. Its aim is that of bringing the system to a domain configuration where the execution can be resumed.
- The third refer to the need for fulfilling a compensation goal labeling an activity. In this case the ***compensation mechanism*** is applied. It allows the system to dynamically provide a context-aware compensation process that is simply a composition of fragments selected by taking into account the current context and whose execution fulfills the compensation goal.

Furthermore, these mechanisms can also be combined together by realizing *adaptation strategies*. These allow the system to handle more complex adaptation needs. These mechanisms and strategies have all been implemented in an adaptation engine able to manage them [121].

Eventually, the platform provider is in charge of defining the adaptation mechanisms and techniques used in the platform. The platform provider

must also provide to the platform's users a way to use all the required adaptation tools, to allow them to understand and exploit the adaptation techniques, when defining their applications on top of the platform (see Figure 3.3 – label B1). In conclusion, we want to highlight here that other adaptation approaches can be exploited in our framework, as an alternative or in addition to the AI-based planning approach that we actually use.

Service providers view. Different service providers can exploit the platform for defining their own applications or simply new value-added services. To this aim, service providers have just to configure the adaptation mechanisms provided by the platform in the specific domain of their services and/or applications (see Figure 3.3 – label B2). In other words, service providers can annotate the processes in the domain objects, by making their applications compliant with respect to the domain model they deal with and to the functionalities that their applications are expected to perform. As a result, service providers will be able to release adaptive service-based applications that can be customized and executed on top on the platform.

End-users view. The end-users finally make use of the available adaptive applications. They are those effectively enacting the adaptation techniques (see Figure 3.3 – label B3). Indeed, adapted application instances are dynamically created, customized and run over their requirements, based on their applications usage. This happens thanks to the adaptation mechanisms (e.g., local and refinement) that are effectively triggered, and then performed, during the users interactions with the applications. Once the specific user execution environment is known, appropriate fragments (i.e., services) can be selected, composed and exploited to satisfy the different user's goals. At the same time, exogenous changes are also managed and solved taking into account the specific execution environment.

Moreover, an important aspect of our approach, which will be detailed and analyzed in chapter 5, is the capability of domain objects to dynamically extend their knowledge on the execution environment, when executed, thanks to the exchange of fragments with other domain objects, allowing them to discover new domain concepts (e.g., domain properties in the domain model).

3.2.3 The Interaction Perspective

Our approach sees the involvement of three main actors, the platform provider, service providers and end-users. Each of them is differently involved in the development, operation and usage of the system and they differently interact with it (see the interaction level in Figure 3.2).

Platform provider view. The platform provider, together with his team, is in charge of realizing the platform and its enablers, and then using it to realize and provide different adaptive service-based systems or simply adaptive services. In order to allow external service providers to exploit these systems through the platform, for turning them into new value-added services and/or applications, the platform provider makes available all the tools, the modeling environment and languages, the access to the different engines running in the platform, through an access console that can be realized in different ways, by using different technologies.

Service providers view. The service providers perspective, play a double role. From one side, they act as platform users. Indeed, they use the platform (i.e., its tools, enablers, engines, services) as a third-party provided service. They can exploit all the services modeled as domain objects and stored in the system knowledge base, modeling both domain-specific and general purpose services in a given domain, to develop their own value-added services. To this aim, service providers access to and interact with the platform's console. From another side, service providers can decide to release their value-added services as systems and/or applications. To this aim and from an interaction point of view, they can decide about the technologies to use for developing their applications (e.g., mobile apps, web applications) and also define the corresponding user interfaces. While for the back-end of their applications service providers exploit the platform, for the front-end they are independent from the platform and its console.

End-users view. The end-users, finally, are those effectively using the different applications developed on top of the platform. They are not aware of the platform itself and exploit it in a completely transparent way. End-users just interact with the available applications through their interfaces, also using different devices, such as their smartphones, laptop, tablet and

so on, depending on the specific technologies through which the service providers released their applications.

3.3 Discussion

After the definition of the whole life-cycle of our design for adaptation approach, we can recognize that realizing adaptive service-based systems, applications, or simply value-added services, through the combination of functionalities provided by the available services in a given domain, demand mechanisms enabling the adaptation, introduced both in the design and in the runtime phases. Indeed, we have seen why pre-defining all the possible combinations and interactions among services is of dubious interest in open and dynamic environments. Since these environments are characterized by continuous changes, such as the entrance or leaving of services, changes in the behavior of these services, changes in the environment itself, involvement of proactive users and so on, every pre-defined behavior or procedure of a system is destined to become obsolete or inapplicable as soon as the context changes. Moreover, a user's goal can be reached through different *composite services*, which are not always available, since they strictly depend on the availability of the *component services* making them. The intuition behind our approach is that of defining abstract procedures to describe the behavior of a system while leaving their customization and context-aware specification as close as possible to the run-time execution of the system. In such a way, the concrete specification of procedures can consider the most up-to-date version of their execution environment, by fully avoiding of being obsolete or inapplicable because of the unavailability of services.

In this context, applicable approaches are those giving to service providers the possibility of defining *dynamically customizable behaviors* for their systems, from one side, and a *dynamic management of the services* exploited in the specific domain and their interactions, from another side, in order to reflect the dynamicity and the openness of the environment in which systems live. However, this is challenging to be realized and it also requires a high degree of automation.

In this direction, the approach that we propose in this dissertation is characterized by the following key aspects:

- it implements the separation of concerns (adaptation vs. application logic) by using (1) the *domain model* to abstract the domain concepts and to specify, on top of it, rich composition requirements, and (2) the *domain objects model* for the uniform specifications of the autonomous and heterogeneous services and their dynamic interaction. The domain objects model also facilitates and allows for the *continuous development* and deployment of new services, as domain objects, that can be easily integrated in the existing systems, to face the openness and dynamicity of the environment;

- through the exploited models, the approach allows for the specification of services behaviors and functionalities as processes, with the additional possibility of defining *dynamically customizable processes*, that can be concretely specified at run-time, to guarantee the context-aware execution of systems;

- for the context-aware customization of processes and the run-time adaptation of systems in case of exogenous changes, the approach exploits automated planning techniques for the dynamic and incremental service composition [25], which perform well in dynamic environments.

The approach that we are going to present in this dissertation, is a proposal to solve the previous open issues and to provide a complete solution for services management and exploitation. Thanks to this general approach, we can facilitate services integration and interoperability, thus better exploiting their functionalities and meeting the user needs. By following our approach, a novel ecosystem of customizable services that are easily personalized in different contexts and user needs, can be designed, deployed, adapted and made available to the interested stakeholders that want to use them to create new services and applications. Indeed, it offers a lightweight-model, with respect to the existing languages for service modeling and adaptation, and it can be implemented with every object-oriented languages.

In the next chapter we describe and formalize the models for designing adaptive service-based systems, while chapter 5 is devoted to show their execution and adaptation in action.

Chapter 4

Adaptive Service-Based Systems: Modeling

This chapter is devoted to the modeling aspects of adaptive service-based systems. As introduced in chapter 3, one of the intuitions behind the design for adaptation approach, proposed in this dissertation, is that of introducing adaptation mechanisms in the life-cycle of systems, by starting from the design phase. Our opinion, indeed, is that designing systems with the adaptation in mind is the only way for making *adaptive by design* systems, by avoiding to handle the adaptation as an exception. Of course, this must be supported by the modeling approach.

The central idea of our approach is the use of two separate models, namely the *domain model* and the *domain objects model*, implementing the separation of concerns principle (*adaptation* vs. *application* logic). The domain model describes the domain of the specific service-based system (e.g., mobility), by defining its concepts (e.g., journey planning, bus journey, ride-sharing journey, ticket payment, etc). The domain objects model, instead, allows the specification of services as concrete implementations of the domain concepts (e.g., existing bus company services, such as Flixbus [1], existing journey planners, such as Rome2Rio [2]). Keeping the two models separate allows the conceptual definition of the operational semantic of services (i.e., in the domain model), over which composition/adaptation requirements can be expressed, by detaching this semantic from the different implementations that might be provided by a plethora of different services (i.e., in the domain objects model). Moreover, by taking advan-

[1] https://www.flixbus.com/ [2] https://www.rome2rio.com/

tage of the separation of models, the whole approach provides support for the definition of *dynamically customizable services*. This allows for the context-aware customization of services in specific domains, through the dynamic and automatic discovery and selection of the better functionalities (or *composition* of functionalities) to be exploited.

The following chapter is mainly based on the work in [27], where the modeling approach was firstly introduced, and on the work in [28], where it was further formalized. We build on top of this work and we further extend it. We start with an overview on the general framework and its models with their core elements, in section 4.1. Afterwards we give formal definitions of the models elements, in section 4.2. Chapter 5, instead, will be devoted to the *runtime* perspective.

4.1 General Framework and Approach

In this section we present the *design* side of our approach for modeling and executing adaptive service-based systems. We informally describe the core elements of the design for adaptation models (i.e., domain objects model and domain model) and give examples on how they might be used to model the *Travel Assistant* described in section 3.1. We start by giving an overview of the travel assistant system in section 4.1.1, to look at the different services and service providers involved, in order to give to the reader an idea of the system complexity and dynamicity. Then, we introduce the general approach in section 4.1.2.

4.1.1 Travel Assistant: a System Overview

As introduced in section 3.1, the scenario that will drive us throughout this dissertation comes from the mobility domain and it concerns with the management and operation of mobility services, within a smart city as well as among different cities/countries. It foresees the involvement of different types of mobility services, from *journey planners*, to *specific transportation means services*, until *shared mobility services*, each referring both to *public* or *private* providers. General-purpose services are also part of the mobility domain, such as *online payment services* that are often

required for the online booking of travel tickets. Furthermore, mobility services are provided via different technologies (e.g., web pages, mobile applications), and they can offer disparate functionalities (e.g., journey planning, booking, online ticket payment, etc), made available through diverse procedures. From a user perspective, they must often look for and interact with different applications to exploit the different functionalities required to organize and accomplish a journey (e.g., a journey planner for planning, one or more mobility services for booking, etc).

The goal of the travel assistant is that of supporting users in the overall organization and execution of their journeys, by supporting the synergistic cooperation among the different mobility services.

However, if the aim is to deliver "smart mobility services" to users, these services cannot be operated each by itself, but should become part of an integrated mobility solution, the *travel assistant system.* This system supports service users (citizens, tourists) and providers (municipality, drivers, transportation companies) in their daily operation and management of the different mobility services. Moreover, the travel assistant also discharges users from looking for specific applications for each specific need, by allowing for the interaction with only one system, released with whatever technology (e.g., mobile app). Such a system *transparently* and *dynamically* discovers, selects and combines the appropriate services for the specific users requests. In addition, the travel assistant should address the requirements listed in section 3.1. Among them, we remark that it should be *generic* enough to guarantee its *portability* among different cities. This would allows the system to be used everywhere a user goes, by following her while moving around.

Figure 4.1 presents a partial and conceptual overview of the different services and service providers in the system, restricting its scope to a subset of the potentially provided mobility services (i.e., travel assistant service, ride sharing service, bus service).

Example. *If we consider the part of the system related to the travel assistant, we notice that the management of this service needs to handle the registration of users (i.e., passengers, drivers, etc) and their access to the system (*`User Profile Management` *component), the planning and organization*

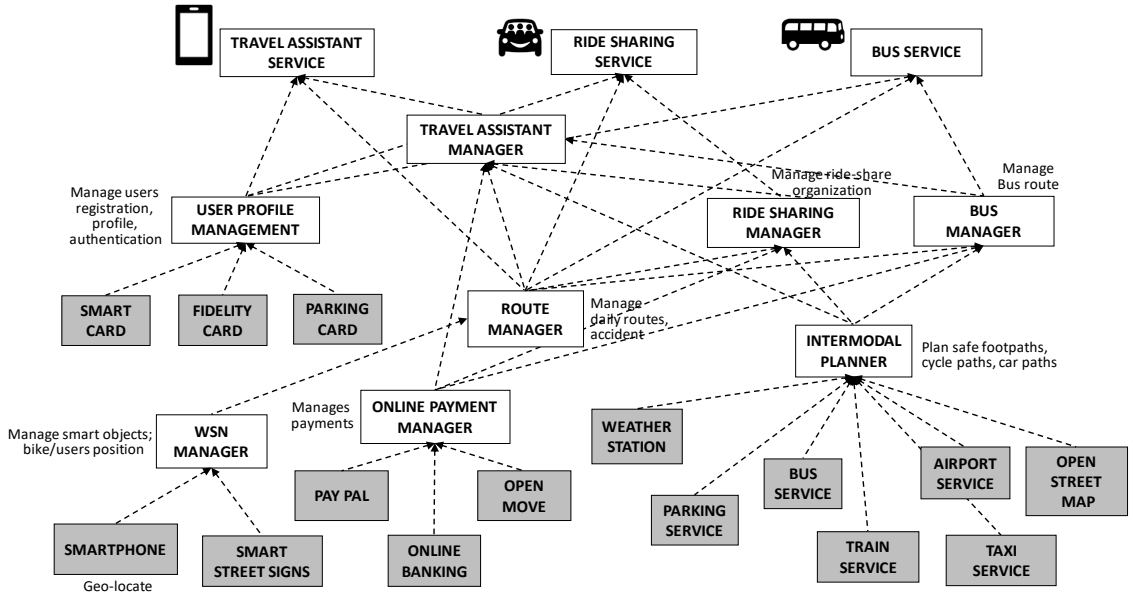


Figure 4.1: Travel assistant: a partial overview of the system.

of journeys (`Intermodal Planner` component) and the organization of routes (`Route Manager` component) taking into account the needs of users but also the route safety (e.g., presence of sidewalks, traffic situation). The daily operation of the service requires to handle journey planning requests from users and the mobility offers from cities, private users and transportation companies (`Travel Assistant Manager` component), the confirmation of journeys and the tickets payment (`Online Payment Manager` component), the tracking of transport means and passengers position (`Wireless Sensor Network (WSN) Manager` component), as well as managing possible exceptions and changes (e.g. find a substitute for a passenger in a ride-share journey, change the bus route due to roadworks, suspend the route due to traffic conditions). Some of these services might be in common with the ride-sharing and bus services (e.g. `Route Manager`, `User Profile Management` components). This would allow not only to avoid replication, but also to enable synergies and collaboration among the different services (e.g., exploit the ride-sharing service as an alternative to the bus in case of a strike).

A first characteristic of the system that clearly emerges from this example, is the variety and heterogeneity of services that the travel assistant

needs to cope with: from *domain-specific* functionalities related to the mobility domain (e.g., management of bus, train and car routes, support for intermodal planning) to *general-purpose* ones (e.g., user profile and access management, user tracking); from *back-end* functionalities (in grey in the Figure 4.1) that need to interact with third-party systems and devices (e.g. online payment of tickets, interacting with smart objects and sensors) to *front-end* ones (e.g., mobile and Web apps to be accessed by the different users). Moreover, the travel assistant needs to deal with the dynamicity of the scenario, both in terms of the *variability* of the actors and services involved, and of the context changes affecting its operation. In particular, the system should be open and extensible, which means that new services (e.g., a new bike-sharing service, a new ticket payment service, a new tracking device), as well as changes in existing services (e.g., changes in a service login procedure, changes in any third-party system) should be easily managed and require minimum maintenance. This is made particularly challenging by the collective nature of some mobility services (e.g., ride-sharing service) to be provisioned, since their operation requires the interaction and collaboration of different autonomous actors (drivers, passengers), and thus results in a high degree of connection and inter-dependencies among the different system services (as shown by dotted arrows in Figure 4.1).

In the next section we will show how the design for adaptation approach proposed in this dissertation addresses these challenges and enables the development of an adaptive by design travel assistant.

4.1.2 The Design for Adaptation Approach

In this section, we introduce the *domain model* and the *domain objects model*. As introduced in chapter 3, the former describes the operational environment of the system, in a given domain, while the latter gives a uniform representation of the autonomous and heterogeneous services making the system, each implementing a specific concept of the domain model. Moreover, through the chapter we provide a mapping between each new introduced element and a corresponding example within the travel assistant system, to help the reader in understanding the models and their interconnections.

The travel assistant system is modeled through a set of *domain objects* representing the services of the system (e.g. `Travel Assistant`, `Journey Manager`). As depicted in Figure 4.2, each domain object is characterized by a *core process*, implementing its own behavior, and a set of process *fragments*, representing the functionalities it provides.

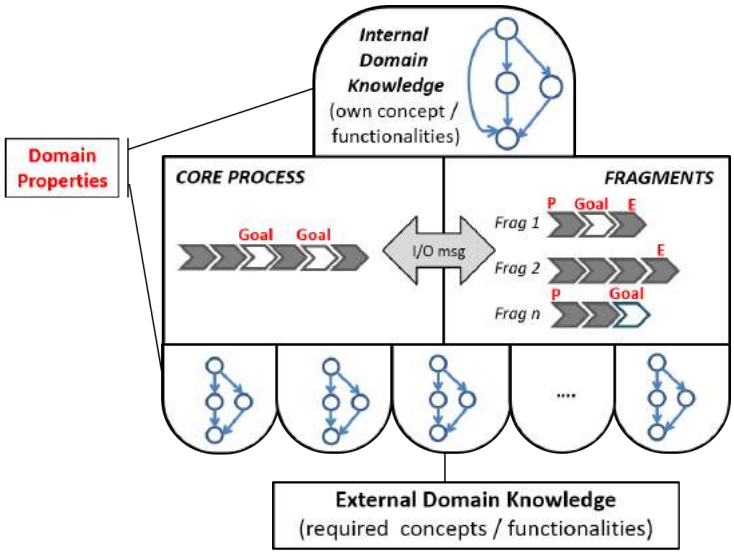


Figure 4.2: Domain Object Model.

Fragments [110, 122] are executable processes that can be received and executed by other domain objects to exploit a specific functionality of the provider domain object. Exposed fragments and the core process communicate through the execution of input/output activities. This concerns the fact that fragments act as an interface for the internal behavior of a domain object, thus they need to interact with the core process to eventually accomplish the functionalities they model.

Unlike traditional system specifications, where services' behavior are completely specified pre-deployment, our approach allows the partial specification of the expected operation of domain objects through ***abstract activities*** that are specified in terms of the desired behavior (i.e., goal) and are *refined at run-time* according to the fragments offered by the other domain objects in the system.

We illustrate this notion with a simple example.

Example. *For instance, in Figure 4.3 we show a portion of the travel assistant system made by a subset of its services and their potential dependencies. Within this portion of the system, the* `Journey Planners Manager` *domain object can partially define the functionality allowing the planning of a journey. Then, different journey planners can join the system and publish different planning procedures, covering areas of varying size and boundaries (i.e.,* local *and* global *journey planners). Only at run-time, when the user's provided source and destination points are known, (e.g. they may belong to the same city, to different cities, to different countries), the* `Journey Planners Manager` *will discover the fragments offered by the specific domain object modeling the most appropriate journey planner for the specified input, and it will use these fragments to refine its abstract activity and to eventually get the list of available multi-modal alternatives.*

An important aspect of the design model that strongly supports the system's dynamicity consists in the fact that ***abstract activities can be used in the core process of a domain object as well as in the fragments it provides***.

In the first case, the domain object leaves under-specified some activities, in his own behavior, that are automatically refined at run-time. The latter case is more complex, and enables a *higher level of dynamicity*, since it allows a domain object to expose a partially specified fragment whose execution does not rely only on communications with its core process but also on fragments provided by other domain objects, thus enabling a *chain of refinements*.

These dynamic features offered by the framework rely on a set of *domain concepts* describing the operational environment of the system, on which each domain object has a partial view.

In particular (see Figure 4.2), the ***internal domain knowledge*** captures the behavior of the domain concept implemented by the domain object, while the ***external domain knowledge*** represents domain concepts that are required to accomplish its behavior but for whose implementation it relies on other domain objects. Domain knowledge

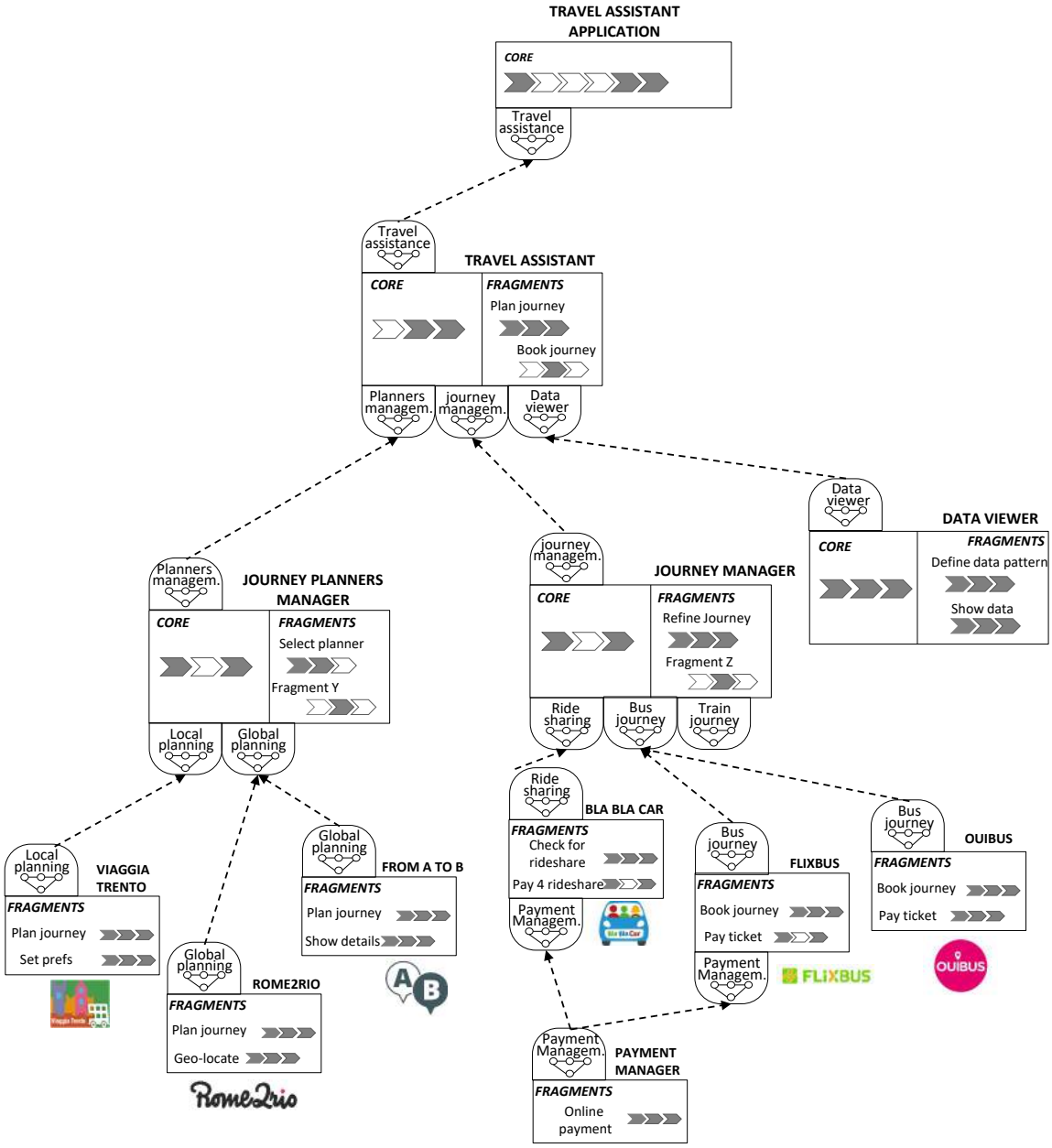


Figure 4.3: Portion of the travel assistant system.

(both internal and external) is defined through *domain properties*, each giving a high-level representation of a domain concept (e.g. journey planning, ride-sharing journey).

Domain properties are modeled as state transition systems and their evolution represent the behavior of the domain concept that they model. At

this point we must clarify that even if in Figure 4.2 we show the domain properties as part of domain objects, which is actually true, we say that domain properties exist independently of the domain objects implementing or relying on them, if any. Indeed, they are identified and defined by domain experts before the system is developed (i.e., before domain objects are designed) and they make the before-mentioned domain model. For simplifying the graphical representation of complex systems made by different interconnected domain objects, through this dissertation we draw domain properties as part of domain objects.

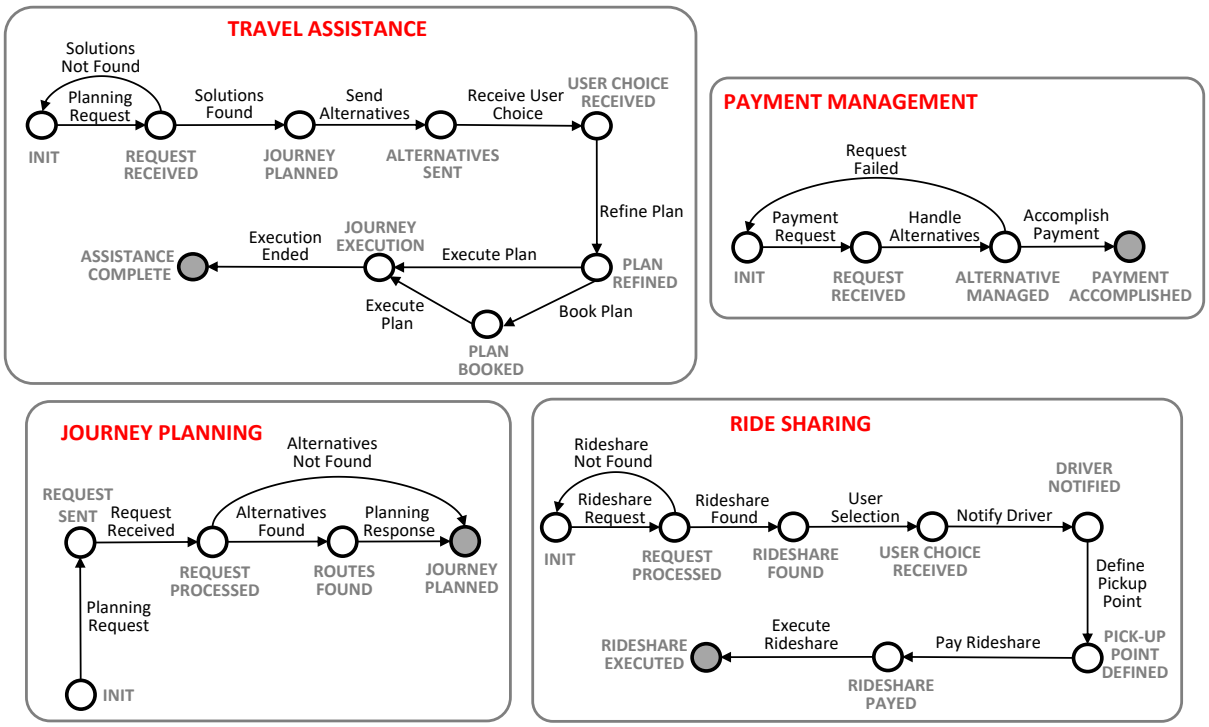


Figure 4.4: Domain properties modeled as state transition systems.

In Figure 7.15 we provide some examples of (simplified) domain properties.

Example. *Consider for instance the domain property* `Travel Assistance`*, which models the typical behavior of a travel assistant used to plan journeys, to let the user select her preferred alternatives, to book it, if required, and finally to execute the journey, all by using the mobility services offered by the travel assistant. First of all, the journey needs to be planned (*`JOURNEY PLANNED`

*state), after that a specific request from the user arrives (*REQUEST RECEIVED *state). Then, the user receives the list of possible alternatives (*ALTERNATIVES SENT *state) and she chooses the preferred solutions among them (*USER CHOICE RECEIVED *state). At this point her plan can be further refined by considering the transportation means effectively composing the chosen alternative (*PLAN REFINED *state). If required by the specific transportation means involved in the user's choice, the plan can be also booked (*PLAN BOOKED *state), otherwise the user can start her journey (*JOURNEY EXECUTION *state) until she reaches her destination (*ASSISTANCE COMPLETE *state). During the normal behavior of the application, a domain property may evolve as an effect of the execution of a fragment activity (e.g., if the journey planning activity goes well, the travel assistant moves in the state* JOURNEY PLANNED*). Otherwise, if something unexpected occurs, a domain property may also evolve as a result of exogenous changes (e.g., because of roadworks the bus is not passing). Eventually, a domain configuration is given by a snapshot of the domain at a specific time of the journey, capturing the current status of all its domain properties.*

The link between the domain model and the domain objects model is given by ***annotations*** labeling activities in processes and fragments.

Indeed, annotations represent domain-related information and they implicitly define a mapping between the execution of processes and fragments and corresponding changes in the status of domain properties. Furthermore, annotations are quite intuitive because they simply reflect the functional properties of the services in the specific application domain. We remark here that thanks to the use of annotations as a link between an abstract concept (i.e., a domain property) and (one or more of) its implementations, it results easy to modify a scenario to consider new and different service implementations. Indeed, this can be done by properly annotating services (i.e., processes in domain objects) without the need of changing the domain properties and the control-flow requirements defined over them.

Annotations can be of different types. In particular, each abstract activity is defined in terms of the *goal* it needs to achieve, expressed as domain knowledge states to be reached, and it is automatically refined at run time, considering (1) the set of fragments currently provided by other domain objects, (2) the current domain knowledge configuration, and (3) the goal to

be reached. In particular, goals are defined over the external domain knowledge, since they refer to functionalities which belongs to domain properties implemented by other domain objects. They can be defined as disjunctions of conjunctions over states of domain properties, as we will see further on.

Example. *For instance, in Figure 4.5, we report an example of a fragment modeling the functionality of paying for a rideshare (`Rideshare Payment` fragment), as it might be exposed by a ride-sharing mobility services, such as BlaBlaCar* [3]*. The activity `Pay for rideshare` is an abstract activity, represented with a dotted line, labeled with the goal `G1: PAYMENT MANAGEMENT = PAYMENT ACCOMPLISHED` that is defined over the `Payment Management` domain property. Indeed, even if the BlaBlaCar service provides the functionality for paying online for the agreed ride-share, it does not implement it by itself, but it relies on typical payment services for the secure payment over internet.*

In addition to goal annotations, activities in processes and fragments are annotated with *preconditions* and *effects*. Preconditions constrain the activity execution to specific domain knowledge configurations.

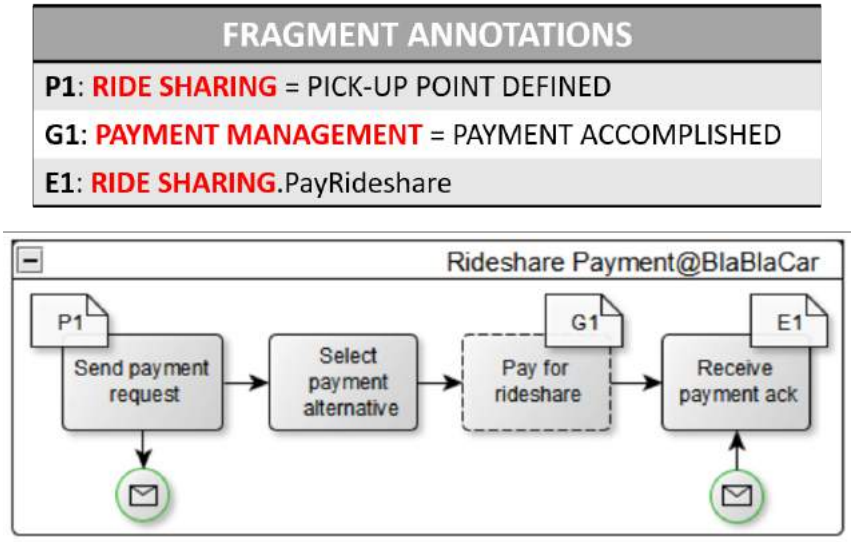


Figure 4.5: Example of a fragment modeling the functionality of paying for a ride-share, as exposed by a ride-sharing mobility services (e.g., BlaBlaCar).

[3] https://www.blablacar.it/

Example. *For instance, in Figure 4.5, the precondition* `P1: RIDE SHARING = PICK-UP POINT DEFINED` *says that, to execute the fragment* `Rideshare Payment`*, the domain property* `RIDE SHARING` *(see Figure 7.15) must be in the state* `PICK-UP POINT DEFINED`*. This precondition constrains the execution of the* `Rideshare Payment` *fragment only in those configurations in which the driver and the passenger already agreed on the details of the ride-share and they defined the pick-up point.*

Effects, indeed, model the expected impact of the activity execution on the domain and represent its evolution in terms of domain properties events.

Example. *For instance, the effect* `E1: RIDE SHARING.PayRideshare` *in Figure 4.5, models the evolution of the* `RIDE SHARING` *domain property (see Figure 7.15). It is caused by the event* `PayRideshare`*, which is triggered by the* `Receive payment ack` *activity in the* `Rideshare Payment` *fragment (in Figue 4.5) of the* `BlaBlaCar` *domain object, and it brings the property in the state* `RIDESHARE PAYED`*.*

Preconditions and effects are used to model how the execution of fragments is constrained by and evolve the domain knowledge. This information is used to identify the fragment (or composition of fragments) that can be used to refine an abstract activity in a specific domain knowledge configuration.

Example. *For instance, as shown in Figure 4.3, the* `RIDE SHARING` *domain property belongs to the internal domain knowledge of the* `BlaBlaCar` *domain object and to the external domain knowledge of the* `Journey Manager`*. This property can be used to specify goals of abstract activities within the* `Journey Manager` *(e.g. to handle a ride-share journey). Similarly, fragments offered by the* `BlaBlaCar` *domain object are annotated with preconditions and effects on the* `RIDE SHARING` *domain property, as just seen for the fragment* `Rideshare Payment`*.*

Potential dependencies (***soft dependencies***, from here on) are established between a domain object and all those domain objects in the system whose modeled domain concept (internal domain knowledge)

matches with one of its required behaviors (domain property in its external domain knowledge).

Figure 4.3 shows the soft dependencies (dashed arrows) among some of the domain objects related to the travel assistant system. A soft dependency between two domain objects becomes a *strong dependency* if, during the system execution, they inter-operate by exchanging their fragments and domain knowledge. In the following chapter on the execution of service-based systems, we present a run-time scenario and show how some of these dependencies become strong dependencies (represented as solid arrows) after the refinement of abstract activities.

Eventually, the resulting adaptive system can be seen as a dynamic network of interconnected domain objects which dynamically inter-operate. In particular, the network takes the form of a *hierarchy of domain objects*, where the abstract activities refinement mechanism enables a *bottom-up approach* allowing fragments, once they are selected for the composition, to climb the domain objects' hierarchy to be injected in the running processes. Within the hierarchy, there may be essentially three kinds of domain objects: *leafs* domain objects model service consumers that do not expose any functionalities but need only to exploit other domain objects fragments to refine their processes (e.g. *Travel Assistant App* in Figure 4.3); *roots* domain objects model pure service providers by offering functionalities and without requiring external knowledge (e.g. *BlaBlaCar* in Figure 4.3); *generic* domain objects model, instead, act both as consumers and providers, so they have both the internal and external domain knowledge as well as abstract activities in their processes (e.g. *Journey Planners Manager* in Figure 4.3).

Notice that the external domain knowledge of a domain object is not static since, if during a refinement a domain object injects in its own core process a fragment containing abstract activities, it receives also the domain properties on which the fragment execution relies on, thus spanning its external knowledge. This dynamicity is reflected in the soft dependencies between domain objects because new dependencies might be established due to refinements, as we will better see in chapter 5. This is due to the fact that the system dynamically evolves at run-time either for the

entrance/exit of a domain object (change in the soft-dependencies), for a change in the fragments it provides (again, potential change in the soft-dependencies), or as a result of an abstract activity refinement (change in the concrete dependencies and, in case of a refinement in a fragment, also in the soft dependencies of the system).

Having introduced all the ingredients of the design for adaptation models, in the next section we provide their formalization.

4.2 Models Formalization

In this section, we give formal definitions of the core elements of our approach, informally introduced in the previous section. Firstly we define the *domain model* in section 4.2.1 and then we formalize the *domain objects model* elements in section 4.2.2. We refer, instead, to chapter 5 for the execution model formalization of systems designed by using our approach.

4.2.1 Domain Model

In this section we formalize the domain model through the definition of the domain property concept as its founding element.

Definition 1 (Domain Property)**.** *A* domain property *is a state transition system* $dp = \langle L, l^0, E, T \rangle$*, where:* L *is a set of states and* $l^0 \in L$ *is the initial state;* E *is a set of events; and* $T \subseteq L \times E \times L$ *is a transition relation.*

We denote with $L(dp)$, $E(dp)$, $T(dp)$ the corresponding elements of dp.

We have previously shown examples of domain properties in Figure 7.15 where, for each domain property, the initial state is denoted by the name INIT, transitions are represented as arrows among states, and events are given by the labels annotating the transitions.

Definition 2 (Domain model)**.** *A* domain model *is a set of domain properties* $C = \{dp_1, dp_2, \ldots, dp_n\}$ *with* $dp_i = \langle L_i, l_i^0, E_i, T_i \rangle$ *for every* $1 \leq i \leq n$*, and such that for every pair* $1 \leq i, j \leq n$*, if* $i \neq j$*, then* $E_i \cap E_j = \emptyset$*.*

The set of all domain states is defined as $L_C = \prod_{i=1}^{n} L_i$ *and the initial context state is* $l_C^0 = (l_1^0, l_2^0, \ldots, l_n^0)$. *The set of all domain events is* $E_C = \bigcup_{i=1}^{n} E_i$. *Finally, the transition relation in the domain model is given as* T_C *such that for every pair of states* $(l_1, \ldots, l_n) \in L_C$ *and* $(l'_1, \ldots, l'_n) \in L_C$, *and for every event* $e \in E_C$, *if* $e \in E_i$ *then* $((l_1, \ldots, l_n), e, (l'_1, \ldots, l'_n)) \in T_C$ *iff*

$$(l_i, e, l'_i) \in T_i, \text{ and for every } j \neq i \text{ we have } l_j = l'_j.$$

A domain model consists in a set of domain properties. We assume that two distinct domain properties $p_i, p_j \in C$ in a domain model do not intersect. The states of a domain model is the product of its domain properties. A state in a domain model can then be seen as the conjunction of states of domain properties. The events of a domain model is the union of the events of its domain properties. Transitions in a domain model are component-wise: each transition changes the state of at most one domain property.

Given a domain model $C = \{dp_1, dp_2, \ldots, dp_n\}$, it will be convenient to denote with $l_i = \bar{l} \downarrow_{dp_i}$ the projection of state $\bar{l} \in L_C$ onto the domain property dp_i.

4.2.2 Domain Objects Model

In this section we start by introducing all the elements that form a domain object, then we show how domain objects combine to form an adaptive system.

The domain models defined previously are instrumental in the definitions of internal and external knowledge of domain objects. We proceed with their definitions.

A domain object will have an internal domain knowledge.

Definition 3 (Internal Domain Knowledge)**.** *An* internal domain knowledge *is a domain model* $\mathbb{DK}_I = \{dp_I\}$ *where* dp_I *is a domain property that represents the domain concept implemented by the domain object.*

For instance, let us consider the FLIXBUS domain object in Figure 4.3. Its internal domain knowledge is given by the singleton containing the BUS JOURNEY domain property.

A domain object will also have an external domain knowledge.

Definition 4 (External Domain Knowledge)**.** *An* external domain knowledge *is a domain model* $\mathbb{DK}_E = \{dp_1, \ldots, dp_n\}$*, where each* dp_i*,* $1 \leq i \leq n$*, are domain properties that the domain object uses for its operation but that are not under its own control.*

For instance, in the FLIXBUS domain object in Figure 4.3, its external domain knowledge is given by the singleton containing the PAYMENT MANAGEMENT domain property, since the Flixbus service requires for the online booking and payment of the tickets, but it does not implements the payment service. Notice that in general, the external knowledge can contain more than one domain property.

The external domain knowledge and the internal domain knowledge are domain models. Hence, they have a set of states, and set of events, and a transition relation as specified in Definition 2. For convenience, we denote $\mathbb{L}_E$ and $\mathbb{E}_E$ the set of states and the set of events in the external domain knowledge. We also denote $\mathbb{L}_I$ and $\mathbb{E}_I$ the set of states and the set of events in the internal domain knowledge.

Both the internal behavior of a domain object, as well as the fragments it provides to others, are modeled as processes. A process is a state transition system, where each transition corresponds to a process activity. In particular, we distinguish four kind of activities: *input* and *output* activities model communications among domain objects; *concrete* activities model internal operations; and *abstract* activities correspond to abstract tasks to be refined at run-time. All activities can be annotated with preconditions and effects, while abstract activities are annotated also with goals. For instance, let consider the example of fragment shown in Figure 4.5: input/output activities are represented with an entering/outgoing message; abstract activities are drawn with a dotted line, while concrete activities are defined by solid lines. We define a *process* as follows:

Definition 5 (Process)**.** *A* process *defined over and internal domain knowledge* $\mathbb{DK}_I$ *and external domain knowledge* $\mathbb{DK}_E$ *is a tuple* $p = \langle S, S_0, A, T,$

Ann$\rangle$, *where:*

- S *is a set of states and* $S_0 \subseteq S$ *is a set of initial states;*
- $A = A_{in} \cup A_{out} \cup A_{con} \cup A_{abs}$ *is a set of activities, where* A_{in} *is a set of input activities,* A_{out} *is a set of output activities,* A_{con} *is a set of concrete activities, and* A_{abs} *is a set of abstract activities.* A_{in}, A_{out}, A_{con}, *and* A_{abs} *are disjoint sets;*
- $T \subseteq S \times A \times S$ *is a transition relation;*
- $Ann = \langle Pre, Eff, Goal \rangle$ *is a process annotation, where* $Pre : A \rightarrow 2^{\mathbb{L}_I} \cup 2^{\mathbb{L}_E}$ *is the precondition labeling function,* $Eff : A \rightarrow 2^{\mathbb{E}_I} \cup 2^{\mathbb{E}_E}$ *is the effect labeling function, and* $Goal : A_{abs} \rightarrow 2^{\mathbb{L}_E}$ *is the goal labeling function.*

We denote with $S(p)$, $A(p)$, and so on, the corresponding elements of p.

We say that the precondition of the activity a is satisfied in the domain knowledge state $\bar{l} \in \mathbb{L}_I \cup \mathbb{L}_E$, and denote it with $\bar{l} \models Pre(a)$, if $\bar{l} \in Pre(a)$. Similarly, we say that the goal of the activity a is satisfied in $\bar{l} \in \mathbb{L}_I \cup \mathbb{L}_E$, and denote it with $\bar{l} \models Goal(a)$, if $\bar{l} \in Goal(a)$. Notice that the goal of an abstract activity specifies a subset of states in the external domain knowledge. As mentioned earlier, a goal can thus effectively be seen as a "disjunction of conjunctions" of states of domain properties. We say that the effects of activity a are applicable in the domain knowledge state $\bar{l} \in \mathbb{L}_I \cup \mathbb{L}_E$, if for each event $e \in Eff(a)$ there exists a $dp_i \in \mathbb{DK}$ and $l'_i \in L(dp_i)$ such that $(\bar{l} \downarrow_{dp_i}, e, l'_i) \in T(dp_i)$.

In particular, in our approach, processes are modeled as Adaptable Pervasive Flows (APF) that is an extension of traditional work-flow languages making processes suitable for adaptation and execution in dynamic environments. The language used to shape processes is the APF language [123], whose syntax corresponds to that defined in Definition 5, while its semantic has been characterized in [123].

Definition 6 (Domain Object)**.** *A* domain object *is a tuple* $o = \langle \mathbb{DK}_I, \mathbb{DK}_E, p, \mathbb{F} \rangle$, *where:*

- $\mathbb{DK}_I$ *is an internal domain knowledge,*
- $\mathbb{DK}_E$ *is an external domain knowledge,*

- *p is a process, called* core process*, defined on* $\mathbb{DK}_I$ *and* $\mathbb{DK}_E$*,*
- $\mathbb{F} = \{f_1, \ldots, f_n\}$ *is a set of processes, called* fragments*, defined on* $\mathbb{DK}_I$ *and* $\mathbb{DK}_E$*, where for each* $f_i \in F$*,* $a \in A_{in}(f_i)$ *implies* $a \in A_{out}(p)$ *and* $a \in A_{out}(f_i)$ *implies* $a \in A_{in}(p)$*.*

The latter constraint on fragments specification concerns the fact that input/output activities in fragments represent explicit communication with the provider domain object. Thus fragments, once received by other domain objects and injected in their own process, start a peer-to-peer communication with the core process of the provider, that implements the required functionality. A graphical representation of a domain object is reported in Figure 4.2.

Definition 7 (Adaptive System)**.** *An* adaptive system *is modeled as a set of domain objects:* $AS = \{o_1, \ldots, o_n\}$*.*

Figure 4.3, for instance, shows a portion of the travel assistant adaptive system. In it, we say that there is a *soft dependency* between objects o_1 and o_2, denoted with $o_1 \dashleftarrow o_2$, if o_1 requires a functionality that is provided by o_2. A soft dependency is formally defined as follows:

Definition 8 (Soft Domain Objects' Dependency)**.** $\forall o_i, o_j \in AS$ *with* $o_i \neq o_j$ *,* $o_i \dashleftarrow o_j$ *if there exists* $dp_E \in \mathbb{DK}_E(o_i)$ *then there exists* $dp_I \in \mathbb{DK}_I(o_j)$ *such that* $dp_E = dp_I$*.*

In Figure 4.3, soft dependencies are made by dotted arrows among domain objects.

4.3 Discussion

In this chapter we have seen at a lower level of detail the two fundamental models of the design for adaptation approach, namely the *domain model* and the *domain objects model*. Their aim is that of enabling the adaptation of service-based systems from their design phase and support it during the run-time execution of these systems. In particular, they accomplish this task by using a specific approach exploiting specific constructs. Firstly, the approach explicitly handle the domain by managing the dynamicity of

services, which can enter or leave the system at any moment. This is due to the use of the *domain model* that provides an abstract representation of the domain concepts, which can be concretized by different services, each giving their own implementation of a specific concept. For instance, coming back to the travel assistant example and considering the domain concept `RIDE SHARING` in Figure 7.15, it is easy to notice that different ride sharing mobility services can adhere to this concept and give their own procedure to implement it (as done by the BlaBlaCar service in Figure 4.3). If a new ride sharing services join the system, it must just implement the `RIDE SHARING` concept in the domain model, without the need of changing it. Furthermore, thanks to annotations defined on top of the domain model and used also to define *goals*, if a goal predicates over the `RIDE SHARING` property (e.g., Goal: `RIDE SHARING = Rideshare Executed`), the entrance of a new service implementing it does not forces a change in the goal. To the contrary, during the execution of the system, if it is necessary to define a service (i.e., fragment) composition to satisfy the specific requirement, the new service can immediately be considered for the composition (e.g., a ride-share journey might be now accomplished by considering not only BlaBlaCar but also the new available ride-sharing service).

Going on, we have seen as the *domain objects model* allows for the uniform modeling of services as domain objects, which are linked to the domain model through annotations and whose execution let the domain evolve. Also this model supports the adaptation, since it allows the design of an ecosystem of *customizable services* that can be easily personalized in different domains. This is made possible thanks to the use of *abstract activities* representing opening points in the definition of processes and fragments, which allow the services to be refined when the domain is known or discovered, and according to its continuous evolution. To support the dynamic inter-operability among services, a flexible connection strategy between domain objects is provided and it is based of *soft dependencies* (see Def. 8) established at design time among domain objects' offered/required functionalities.

At this point of the discussion, we highlight a couple of aspects that are not managed in the current version of our approach, but that leaves open points for future extensions, as we will better discuss in chapter 8. The first

refers to composition requirements, which are expressed in terms of goals on abstract activities defined as disjunctions of conjuctions over states of domain properties. In particular, they reflects functional properties of services allowing the definition of *control-flow* composition requirements. On the other hand, *data-flow* requirements are not actually considered. Although previous work on data-flow requirements in the composition of Web services exist (see for instance [124] and [125]), they have not yet been integrated into the design for adaptation approach. Besides their integration as future work, we also plan to consider an extension of the domain properties in such a way to consider data variables related to context states. The idea is that of enabling the definition of annotations that consider not only the current context states in the domain configuration but also the values of their data variables.

The second aspect, instead, refers to the handling of unexpected events coming from the context and affecting the execution of service-based applications. As we said, indeed, domain properties may evolve both as an effect of the normal execution of service-based applications and because of exogenous events in the operational context. These events are modeled as transitions in the STS shaping domain properties. While the events representing the normal evolution of a domain property are devised by domain experts during the design phase and they are triggered by the execution of activities in processes and fragments, the events modeling unexpected changes in the context are triggered by the operational context and they do not relate to the execution of service processes. In the current version of our approach, we do not deal with the monitoring of context events, while the approach is able to manage the system's adaptation in case they occur. This limitation can be overcome by extending our approach with existing approaches dealing with the monitoring of the evolving contexts in which service-based applications are executed. For instance, relevant approaches can be found in [107]. A comparison among different run-time monitoring approaches, in various domains and for different purposes, is further provided in [126].

In the next chapter we introduce the adaptation mechanisms and strategies *exploited* and *facilitated* by our design for adaptation approach. We define the enablers for the execution and adaptation of service-based sys-

tems (e.g., process engine, adaptation engine, domain objects manager) together with additional features of the approach enabling a high level of dynamicity of service-based systems defined within our framework.

Chapter 5

Adaptive Service-Based Systems: Execution

This chapter is devoted to the *dynamic and automatic adaptation* of service-based systems defined within our framework. In chapter 4, we have described how our design for adaptation approach supports the modeling of service-based systems, by introducing adaptation artifacts already in the system models. In particular, we have seen how the modeling approach based on *domain objects* allows the definition of service functionalities (i.e., fragments) as *abstract procedures* saying *what* might be done, but without specifying *how* to do it, thus leaving the context-aware specification of these procedures to the run-time execution of the system. This is done especially by exploiting the *abstract activity* construct and the possibility to link the execution of these functionalities to an abstract description of the operational environment of the system, namely the *domain model*, on top of which rich *service composition requirements* labeling functionalities can be defined.

From the execution perspective, it is known that the adaptation may take place at different levels of abstraction of service-oriented architecture. Commonly, these levels are (i) the infrastructure level (ii) the service level and (iii) the application level. In our approach, the adaptation of service-based systems concerns with the *application level*, the most abstract one, which commonly deals with services realized as service-based processes. Indeed, as described in chapter 4, domain objects' *core processes* and *fragments* are both modeled as processes.

As previously introduced, the *execution* of service-based systems defined within our design for adaptation framework relies on previous work on the dynamic adaptation of fragment-based and context-aware business processes [25, 120] based on AI planning [47]. In particular, this work performs a *goal-based adaptation* that is notably suitable for the application in open and dynamic environments, differently from rules-based and built-in approaches. Indeed, the application of approaches belonging to these last two categories requires to specify adaptation tactics at design-time, which is inappropriate to deal with dynamic service-based systems.

This chapter is an extended and revised version of the work presented in [28]. In order to comprehensively define the adaptation mechanisms and strategies exploited by our approach, we provide an overview of the work on the dynamic adaptation of fragment-based and context-aware business processes [25, 120] in section 5.1. In section 5.2, instead, we give a description of the enablers of the design for adaptation framework supporting the execution and the adaptation of service-based systems. For illustration purpose and to ease the explanation, we provided a running scenario of the travel assistant example in section 5.3, which also helps us to present additional features of the approach related to the adaptation of systems (e.g., the dynamic knowledge extension). Eventually, the execution model is formalized in section 5.4, and discussed in section 5.5.

5.1 Background on Adaptation Mechanisms and Strategies

Before showing a running example of the domain objects-based travel assistant, we give, in this section, a background of the exploited adaptation mechanisms and strategies. These implement the dynamic adaptation of fragment-based and context-aware business processes proposed in [25, 120], which are in turn based on AI planning [47]. The link between the approach presented in this dissertation and the approaches in [25, 120] is the use of the APFL language [123] to model business processes (i.e., the equivalent of core processes in this dissertation) and fragments. As already seen in chapter 4, APFL gives the possibility to relate the execution of processes

with the system context, through the use of annotations on process activities (e.g., *preconditions* and *effects*). Moreover, APFL adds the use of *abstract activities* labeled with *goals* and acting as open points enabling the customization and adaptation of processes, as we are going to see in this chapter. These aspects allow developers to define flexible processes that are particularly suitable for *adaptation* and *execution* in dynamic environments. In [25, 120] the authors provides mechanisms and strategies for the dynamic adaptation of APFL-based processes. In this section, we briefly describe these mechanisms and strategies, as well as the automated service composition approach based on AI planning that they exploit [47]. An exhaustive example of their description and application in a car logistic scenario can be found in [127].

The adaptation mechanisms that can be used to handle the dynamicity of the environment in which service-based systems operate essentially deal with three types of adaptation needs:

1. The first one refers to the need for refining an abstract activity when executing a process. This is made by triggering the ***refinement mechanism*** whose execution allows the framework to automatically find and compose available fragments in the system, on the basis of the goal of the abstract activity and the current context. As a result, an executable process whose execution guarantees to reach the abstract activity's goal is provided. The refinement mechanism performed at run-time is particularly advantageous since it allows to postpone the selection of fragments to the run-time phase, when the system knows about their availability (e.g., in the mobility domain, the effective availability of mobility services in a specific time and place can be known only at run-time). Furthermore, the fragments composition depends on the current execution context, which is unknown at design time (e.g., a ride-share journey may be complete with passengers and thus it is not usable).

   Example. *As we will see later on in this chapter, an example of abstract activity could refer to the booking of a travel alternative chosen by the user. During the execution the abstract activity is automatically refined by (a composition of) the fragments implementing the booking*

functionality, if any, and provided by the mobility services belonging to the specific travel alternative.

2. The second mechanism refers to the violation of the precondition of an activity that has to be performed. In this case the ***local adaptation mechanism*** is performed. Its aim is that of bringing the system to a context configuration where the execution can be resumed. This requires for a solution helping in re-starting a faulted process. Also in this case the framework generates a composition of fragments whose execution brings the system to a context configuration in which the activity precondition is satisfied.

   Example. *To give an example in the mobility domain, suppose that the fragment for booking a place in a ride-share starts with an activity whose precondition requires that the user is registered to the specific ride-sharing service. If this precondition is not satisfied when the activity must be performed, the framework could provide a composition made by the fragments allowing the user to register for the selected ride-sharing service, in such a way to restart the faulted process and continue with the booking procedure.*

3. Lastly, the framework allows the specification of activity's compensation as goal over the context, in such a way to avoiding the explicit definition of the activities to be executed and to dynamically provide a context-aware compensation process. This process is simply a composition of fragments selected by taking into account the current context and whose execution fulfills the *compensation goal*. In this case, the ***compensation mechanism*** is applied.

   Example. *A typical example in the mobility domain refers to a situation in which a travel ticket refund is needed. The compensation goal associated to the activity must require that a context configuration where the ticket is not booked for the specific travel. In this case, the activity needs to be compensated after its completion and the generated compensation process requires that the ticket for the travel is refund.*

At this point of the discussion about the exploited adaptation mechanisms, is important to highlight that the AI planning which the goal-based adaptation relies on is able to deal with *stateful* and *non-deterministic* services. The framework, indeed, intuitively allows for the representation of relevant concepts (i.e., the domain, activities preconditions and effects) that give the possibility to relate the execution of processes (i.e., services) with the application context. Moreover, as demonstrated in [47], the fragment composition (i.e., a plan) returned by the AI planner as a result to an adaptation problem is correct by construction, that is, if a plan is found, it is guaranteed that its execution allows to reach a situation in which the goal of the adaptation problem is reached. However, dealing with stateful services implies that the planner may also not find a solution to an adaptation problem. For these reasons, adaptation strategies have been designed. Indeed, the mechanisms introduced above can further be combined together by realizing *adaptation strategies* allowing the system to handle more complex adaptation needs (e.g., the failure of an abstract activity refinement). In particular, we mention the following:

- **Re-refinement strategy:** it combines the *compensation* and the *refinement* mechanisms. It handles the situation in which a faulted activity belongs to the refinement of an abstract activity. In this case, this strategy first compensate the activities that have already been executed and that are annotated with a compensation goal, and then it performs a new refinement of the starting abstract activity, by considering now the new context configuration.

- **Backward adaptation strategy:** it combines the *compensation* and *local* mechanisms. It handles the situation in which, given a new context configuration, it is needed to bring back a process instance to a previous activity in the process that may allow for different execution decisions. The compensation is used to roll-back (some of) the activities, while the local adaptation is required for bringing the system to a new configuration satisfying the activity precondition.

At this point, to conclude this overview on the context-aware adaptation of processes, we briefly describe the fragments composition approach

based on AI planning [47]. Essentially, for the automatic resolution of an adaptation problem, according to [47], this is transformed into a *planning problem*, so that planning techniques can be used to solve it.

In Figure 5.1 we give a graphical overview of the approach, as a simplified version of that shown in [47]. Further details can be also found in [25]. The *adaptation problem* is represented by a set of fragments (e.g., PFi), a

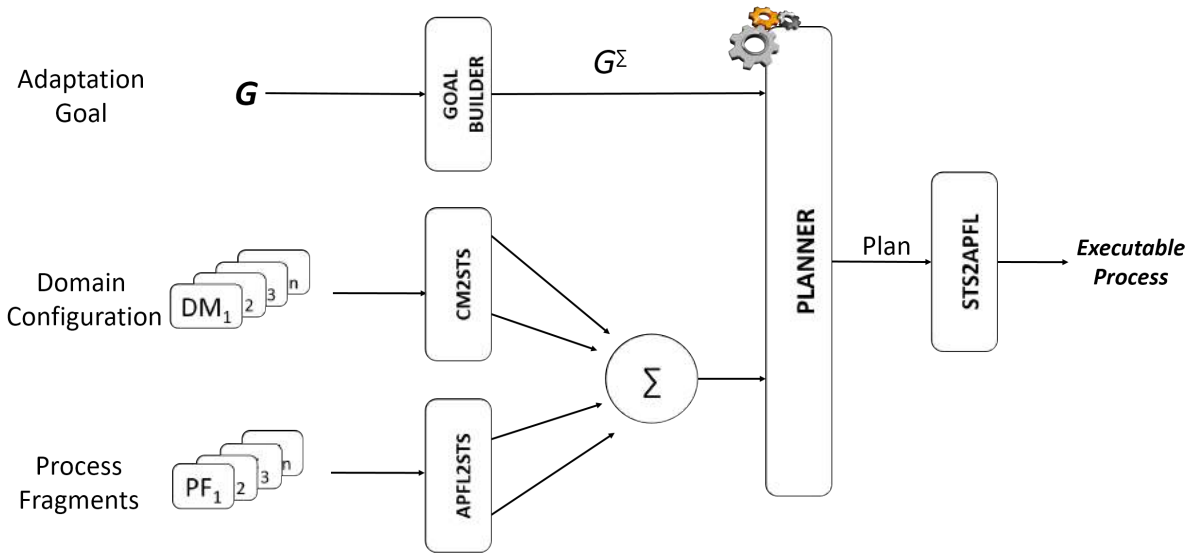


Figure 5.1: Adaptation as AI Planning.

set of domain properties (e.g., DMi) and an adaptation goal. The *planning domain* is then derived from the adaptation problem by transforming fragments and domain properties into STS, by applying transformation rules, such as those presented in [120]. The adaptation goal is, instead, transformed into a set of configuration of the planning domain. At this point, given the specified planning domain, the *planner* generates a *plan* that achieves the adaptation goal. The generated plan is itself an STS, thus it is translated into an executable process that effectively implements the adaptation mechanisms described above.

In conclusion, we remark that the before-mentioned mechanisms and strategies have all been implemented in an **adaptation engine** [121]. This engine is *one* of the enablers of our design for adaptation framework and we exploit it in the way that we will show in the next section.

5.2 Enablers of the Design for Adaptation Framework

To understand the operation of the travel assistant, we first introduce the *execution and adaptation enablers* provided by the design for adaptation framework shown in Figure 5.2. Indeed, the run-time operation of service-based applications defined on top of our platform relies on different enablers supporting their execution and adaptation.

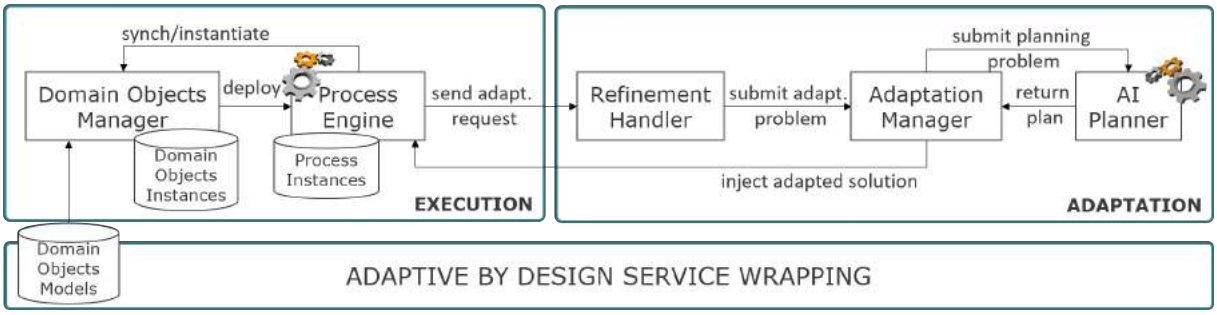


Figure 5.2: Platform Enablers.

The **Execution Enablers**, namely the *Domain Objects Manager* and the *Process Engine*, leverage on the different services wrapped up as domain objects and stored in the system's knowledge base. The execution enablers are in charge of execution the domain objects processes (i.e., core processes and fragments) during the operation of service-based applications. The **Adaptation Enablers**, namely the *Refinement Handler*, the *Adaptation Manager* and the *AI planner*, instead, leverage on the adaptation mechanisms and strategies, described in section 5.1. They are in charge of managing the adaptation needs of applications, arising at run-time. Considered as a whole, they represent the *adaptation engine*.

To start, it is required that developers exploit and wrap up as domain objects the available services in a given domain (e.g., mobility). These are stored in the *Domain Objects Models* repository in Figure 5.2. To understand how the execution and adaptation enablers interact and cooperate, we defined a sequence diagram showing the interaction-flow among them in Figure 5.3.

Domain objects core processes (simply processes from here on) implement the behavior of the services modeled as domain objects. They are ex-

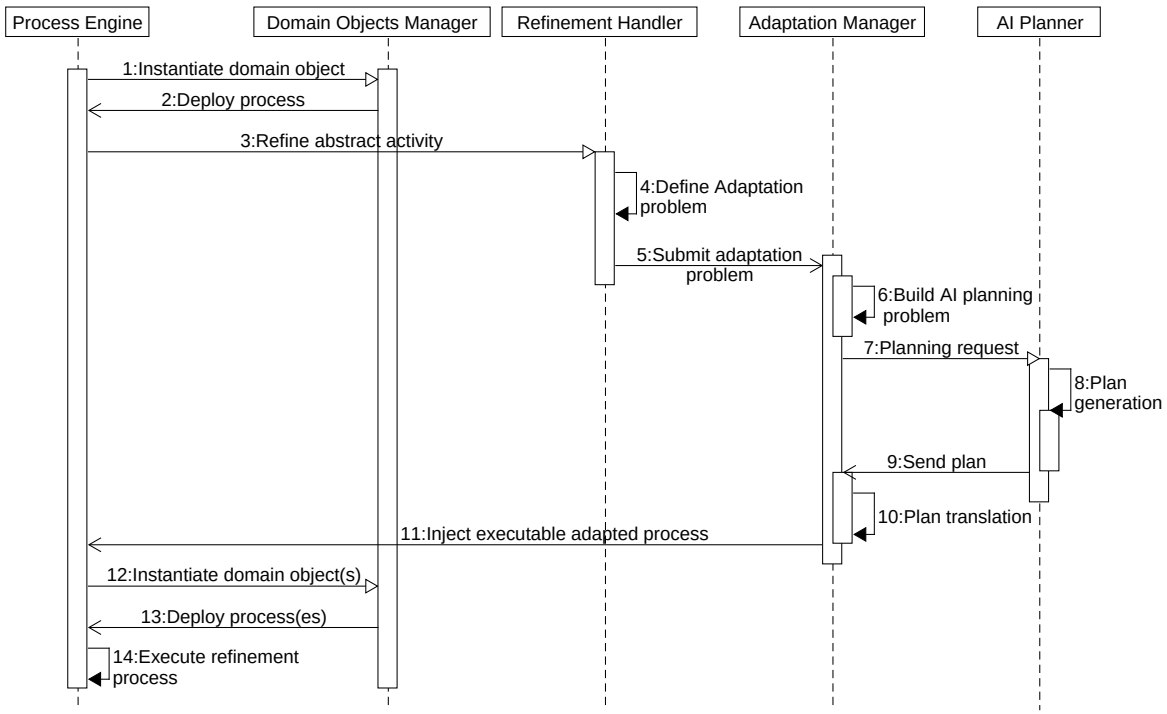


Figure 5.3: Interaction-flow among the execution and adaptation platform enablers.

ecuted by the *Process Engine*. This component is central in our approach: it handles the *decentralized management of processes*, the *communication among processes*, the *dynamic correlation among processes* and the *abstract activity management*, which are needed requirements for the application of our techniques. It manages service requests among processes and, when needed, it sends requests for domain objects instantiation to the *Domain Objects Manager*. A request is sent for each demanded service whose corresponding process has not yet been instantiated. The domain objects manager replies by deploying the requested process on the process engine. In this way, a correlation between the two processes is defined.

During the normal execution of processes, abstract activities can be met. These activities need to be refined with one or a composition of fragments modeling services functionalities. To this aim, the process engine sends a request for abstract activity refinement to the *Refinement Handler* component. This component is in charge of defining the *adaptation problem* corresponding to the received request. Indeed, in our framework

an adaptation need corresponds to the need of refining an abstract activity. An adaptation need is then transformed into a general adaptation problem that contains complete information about the adaptive system instance and an adaptation goal, which captures the adaptation objectives to be fulfilled. The refinement handler implements a pre-planning phase, by defining the *problem domain*. To this aim, it selects the fragments that can potentially be part of the final fragments composition. The selection is driven by the goal defined by the abstract activity. Then, the refinement handler submits the adaptation problem to the *Adaptation Manager*. This translates the *adaptation problem* into a *planning problem* so that it can be solved by the *AI Planner* component. After the plan generation, the AI planner sends the plan to the adaptation manager that will transform it into an executable process. This process can now be sent to the process engine and injected into the abstract activity being refined. At this point, depending on the fragments in the composition, the process engine can request for the instantiation of one or more domain objects, whose processes will be deployed. At the end, the execution of the refinement process can be performed.

We want to highlight, now, that while the adaptation engine implements all the adaptation mechanisms and strategies reported in section 5.1, in our design for adaptation framework we only handle the refinement mechanism. As future work we plan to extend the framework to the management of all the other mechanisms. This is due to the fact that, differently from the adaptation engine, we implemented from scratch both the process engine and the domain objects manager, and we adjusted them for the joint use with the adaptation engine, thus enabling the communication and interoperation with it. This work required a huge implementation effort. Our strategy has been that of focusing on the demonstration of the *feasibility* of our approach, by implementing a prototype of travel assistant exploiting real-world services as domain objects (see chapter 7), while leaving the management of the other adaptation mechanisms for future work. Implementation details will be given in chapter 7.

In the next section we show a concrete example on the running execution of the travel assistant, allowing the readers to better understand the operation of the platform's enablers.

5.3 Travel Assistant: Running Scenario

In this section, we first detail the functioning of the travel assistant and, then, we give an execution example. We have previously shown a (partial) overview of the travel assistant system's model in Figure 4.3 of chapter 4. In particular, the figure shows the *hierarchical* model of the system made by a set of *domain objects* connected through *soft dependencies* among their offered and required functionalities. The focus of this section is, furthermore, that of showing:

- how the domain objects dynamically inter-operate by exchanging and injecting (composition of) fragments, thus enabling a *chain of incremental refinements* (such as that in Figure 5.4). This is due to the use of *abstract activities* in core processes and fragments and to the exploitation of the *refinement mechanism*. Moreover, when two domain objects effectively interact, the soft dependency between them evolves into a *strong dependency*. In other words, their potential dependency emerged at design time really turns into a concrete dependency at run-time;

- how *the refinement process allows domain objects to span their external knowledge on the domain*, by establishing new soft dependency. Indeed, we recall here that the external domain knowledge of a domain object is not static since, if during a refinement a domain object injects in its own core process a fragment containing abstract activities, it receives also the domain properties on which the fragment execution relies on (see section 5.3.1).

Let us now consider the operation of the travel assistant system represented by the `Travel Assistant` domain object in Figure 4.3. Its main features are the following: **(i)** by interacting with the user, it collects her journey requirements (e.g., source and destination points, travel preferences, etc) and it sets up the corresponding journey planning request; **(ii)** given a user journey planning request, it is able to decide between a local or a global planning service to better handle the request, by analyzing the source and destination points entered by the user; **(iii)** given the planner responses,

it defines the better way to show this responses to the user (e.g., a list of travel alternatives, a warning message); **(iv)** once the user selects a specific solution, the travel assistant is able to identify the transport means in the legs making the entire solution. With the information that it gets, it goes vertically to find the proper service(s) to use (e.g., the ones of the specific transport companies), if existing in the system. In this way, it can incrementally provide to the user specific functionalities and context-aware information for her journey. We emphasize that the more (mobility) services are wrapped up and stored in the system's knowledge base, the more responsive and accurate the travel assistant will be.

Executing the travel assistant. In this paragraph, we show the execution of the scenario described above. It represents the situation in which our user, let's say she is called Sara, wants to organize a journey from Trento to Vienna. In Figures 5.4, we report examples of *chains of incremental refinements*, as they are dynamically set up and executed after the specific request of Sara.

We remark that more complex and close to reality alternatives of our scenario can be modeled within our framework. However, even this simple version allows us to highlight the features of the design for adaptation approach and the issues that it tries to address. For presentation purposes and without loss of generality, we report only portions of the processes and fragments involved in the specific scenario. For each fragment, we specify its name and the domain object which it belongs to (e.g., fragmentName@domainObjectName).

We suppose that the travel assistant is provided as a mobile application (modeled by the domain object `Travel Assistant Application` in Figure 4.3), through which Sara is using it. The execution starts from the core process of this mobile app, modeling the user process. Sara starts the process, by starting the application. Then, a sequence of three abstract activities (represented with dotted lines and labeled with a goal) need to be refined. Here we focus on the first one, `Plan Journey`, whose goal models the situation in which Sara ends up with a specific travel plan. To execute this activity, the refinement mechanism is triggered. In the following we list the refinement steps happening while executing the travel assistant and shown in Figure

5.4.

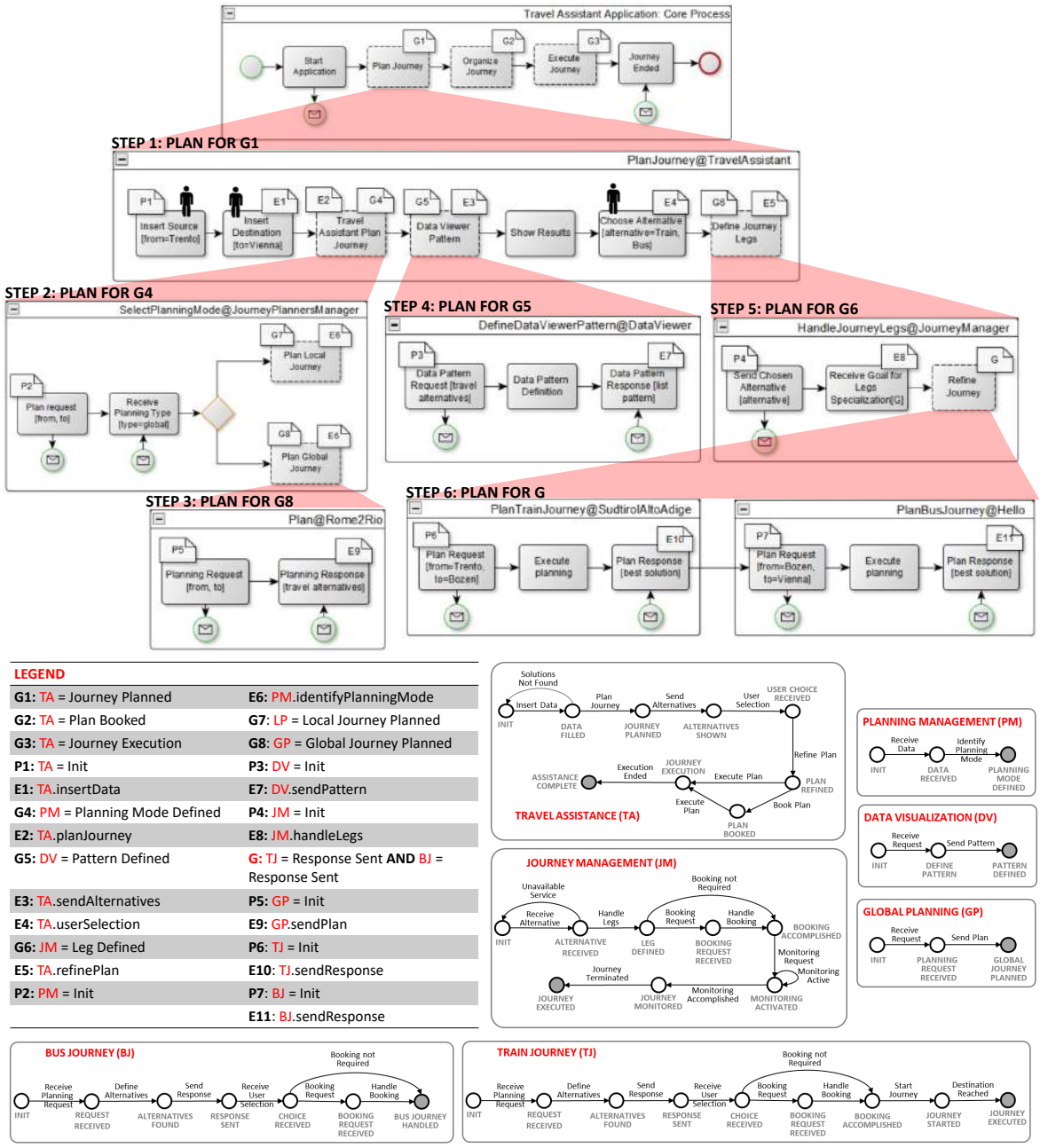


Figure 5.4: A detailed example of the travel assistant execution through incremental and dynamic refinements.

Step 1. The fragment `PlanJourney` provided by the Travel Assistant is selected for the refinement, and injected in the behavior of the mobile app

core process. It implements a wider journey planning functionality, that is from a first planning looking for available alternatives, to a more detailed planning after that a specific alternative has been found and selected by the user. To start, it allows Sara to insert the departure and destination locations.

Step 2. To identify the proper planning mode (*local* vs. *global*), and thus find the better journey planner, the travel assistant domain object relies on the `Planners Management` domain property, as shown by the abstract activity `Travel Assistant Plan Journey` in the `PlanJourney` fragment in execution. The Journey Planners Manager domain object implements the `Planners Management` domain property. Its fragment `SelectPlanningMode` is selected for the refinement. This fragment does not implement any logic, but it needs to communicate with its core process. Indeed, its activities `Plan Request` and `Receive Planning Type` model the communication with its core process, where the request is effectively handled. In particular, the Journey Planners Manager knows only at runtime if a *local* or *global* planning is required. In our scenario, since the locations entered by Sara are Trento and Vienna, the Journey Planners Manager will reply with a global planning type. This will drive the execution of its fragment through the `Plan Global Journey` abstract activity.

Step 3. At this point, one or more fragments provided by the available global journey planners existing as domain objects in the system's knowledge base can be selected for the refinement. In our scenario, we suppose that the `Plan Global Journey` abstract activity is refined with the fragment `Plan` provided by the Rome2Rio domain object, a open global planner service. The execution of this fragment will end up with a list of travel alternatives, if any.

Step 4. After that the chain of incremental refinements made by the steps 1, 2 and 3 has been accomplished, the execution returns to the `PlanJourney` fragment, by continuing with the `DataViewerPattern` abstract activity. Indeed, to properly show the travel alternatives to the user, an appropriate data visualization pattern must be selected, based on the data format (e.g., a list, a message). This is defined at run-time, when the data (and its format) is known. The Data Viewer domain object provides the `DefineDataViewerPattern`

fragment for this purpose. At this point, Sara can receive and visualize on her smartphone the list of the found travel alternatives satisfying her requirements.

Step 5. Sara can now select her preferred alternative (we suppose that she selects a multi-modal solution made by a train and a bus travels). Based on the user choice, the `Define Journey Legs` abstract activity is refined with the `HandleJourneyLegs` fragment provided by the Journey Manager domain object. It is able to dynamically define the goal for the `Refine Journey` abstract activity, that will be `G: TJ = Response Sent AND BJ = Response Sent`, being the selected solutions made by a train and a bus journeys. The refinement of this abstract activity will allows the Travel Assistant system to look for and find the proper fragments for each journey leg. Notice that the `Refine Journey` activity is a so-called *higher order abstract activity*, that we are going to define in the subsequent paragraph.

Step 6. The last step shows a *composition of fragments* provided by the transport companies involved in the legs of the user selection (i.e., Sudtirol Alto Adige and Hello). Their execution provides to Sara the proper solutions, from the two companies, that combined together satisfy her need of planning a journey from Trento to Vienna, passing through Bozen.

Higher Order Abstract Activities. In the *step 5* of the running example, we have presented the `Refine Journey` activity as a *Higher Order Abstract Activity* (HOAA). This kind of activity is actually a regular abstract activity and is managed as such, with the only difference that its goal is defined at execution time, within the fragment or core process it belongs to. For instance, in Figure 5.4 – Step 5, we can notice that the receive activity just preceding the `Refine Journey` HOAA, namely `Receive Goal for Legs Specialization`, is in charge of receiving the HOAA's goal and labeling the HOAA with it, so that, at the next step, the process engine will meet a regular abstract activity to execute.

HOAAs are used when the goal that needs to be specified is totally depending from the run-time execution environment and the only way to specify it, at design time, would be that of pre-defining all the alternatives that is possible to predict. But this is exactly what must be avoided while defining adaptive service-based applications to be executed in dynamic

environments. For this reason, we implemented and introduced in our framework the HOAA construct allowing the dynamic definition of goals (i.e., composition requirements) when the execution domain is known.

Example. *Let us consider the* `HandleJourneyLegs` *fragment at Step 5 in the running example, containing the HOAA. It is exposed by the Journey Manager domain object that is a value-added domain object defined by exploiting and combining the domain objects implementing real-world mobility domain concepts, such as* train journey, bus journey, air journey, *and so on. Indeed, its main task is that of relate the specific travel alternative selected by a user with the proper domain objects able to handle it. At this point, it is easy to notice that a travel solution can be made from any combination of transport means. This implies that the goal of the HOAA, if pre-defined, should model any possible configuration to cover all the corresponding combination of transport means. To the contrary, the Journey Manager domain object implements the logic to dynamically relate each combination of transport means (e.g., train and bus as in our example) with the right goal to be associated with the HOAA (e.g., the goal* `G: TJ = Response Sent AND BJ = Response Sent` *in Figure 5.4) and generates it dynamically. Obviously, the Journey Manager has a view on the system's knowledge base and the available domain objects models, on top of which it runs its logic.*

5.3.1 Dynamic Knowledge Extension

An important feature of our approach that strongly supports the dynamicity of service-based applications is represented by the ability of domain objects to span their knowledge on the whole application domain.

As detailed in chapter 4, each domain object has a partial view on the operational environment of the application, which is represented by *domain properties* in the *domain model*. In particular (see Figure 4.2), while the *internal domain knowledge* in a domain object captures the behavior of the domain concept implemented by the domain object, the *external domain knowledge* is made by domain concepts that are required to accomplish its behavior but for whose implementation it relies on other domain objects.

The *dynamic extension of the knowledge* concerns with the *external domain knowledge* and it is caused by the execution of the abstract activity refinement mechanism. In particular, it happens every time that a domain object injects in its own core process a fragment containing abstract activities. Indeed, since abstract activities are labeled with a goal that predicates on the domain properties in the external knowledge of the domain object which it belongs to, the receiving domain object receives, together with the fragment, also the domain properties on which the fragment execution relies on. These domain properties are going to be part of the external domain knowledge, thus extending it.

We illustrate this feature with a practical example, reported in Figure 5.5. The example is focused on a specific subset of the refinement steps in Figure 5.4. While in 5.4, for a matter of readability of the scenario execution, we show only the fragments involved in each refinement step, in Figure 5.5 we show also (some of) the entire domain objects with their structure, so that we can demonstrate how they evolve, by extending their knowledge. More precisely, the example focuses on what happen to the adaptive travel assistant application instance when the refinement steps 2 and 3 (see Figure 5.4) are performed.

We catch the moment in which Sara is executing the refinement for the goal G1 in the core process of the Travel Assistant Application domain object. She has just entered the departure and destination points, when the process engine meets the `Travel Assistant Plan Journey` abstract activity, whose goal `G4: PM = Planning Mode Defined` is defined over the `Planners Management` domain property in the external knowledge. At this point, the refinement mechanism is triggered and the **Step 2** in Figure 5.5 starts. The adaptation engine replies with the fragment `Select Planning Mode`, which is provided by the Journey Planners Manager domain object, that can thus be injected in place of the abstract activity. Two important things happen right now:

- the dependency between the Travel Assistant Application and the Journey Planners Manager, which was a design time *soft dependency*, becomes a *strong dependency* (represented as a solid arrow) because of the effective inter-operation between the two domain objects due

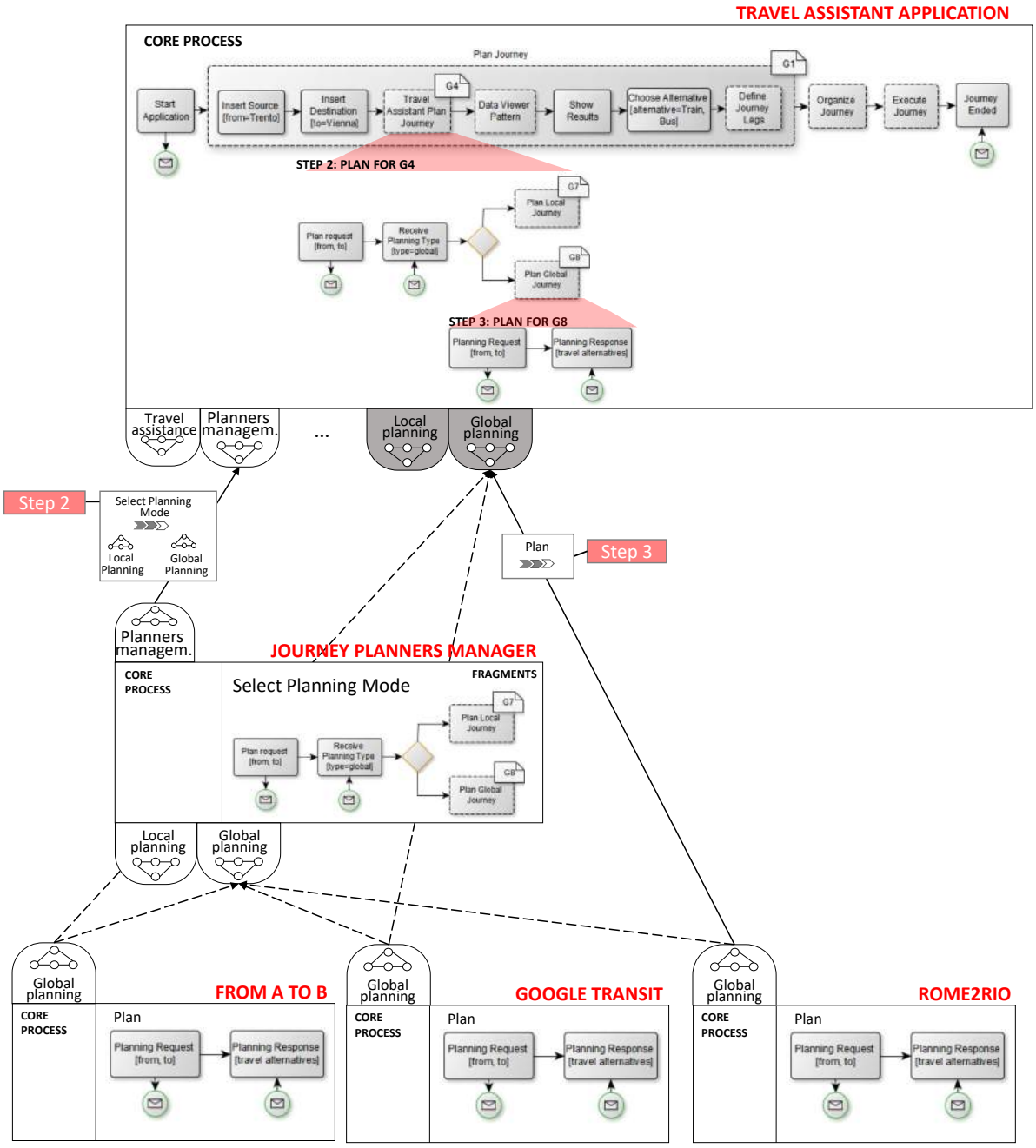


Figure 5.5: Example of the dynamic extension of a domain object's knowledge.

to the exchange of the `Select Planning Mode` fragment;

- since the `Select Planning Mode` fragment is equipped with two abstract activities, namely `Plan Local Journey` and `Plan Global Journey`, labeled with the goals G7 and G8 (see table in Figure 5.4) respectively, the

Travel Assistant Application receives also the `Local Planning` and `Global Planning` domain properties, on which these goals are defined. This dinamicity is now reflected in the soft dependencies of the Travel Assistant Application because new dependencies are established. In particular, it will establish new dependencies with all the domain objects in the system implementing the two just hinerited domain properties. Indeed, by looking at Figure 5.5 and focusing only on the `Global Planning` property, we can see that new soft dependencies exist between the Travel Assistant Application and the global planners currently available, namely *From A to B* [1], *Google Transit* [2] and *Rome2Rio*.

We can notice how the dynamic knowledge extension allows domain objects to dynamically discover new services that they can, in turn, exploit for the refinement of hinerited abstract activities. Indeed, the **step 3** in Figure 5.5 refers to the refinement of the `Plan Global Journey` activity, for which the `Plan` fragment provided by the Rome2Rio domain object is selected. It is easy to note that this refinement would not have been possible without the dynamic extension of the knowledge because, in its design time version, the Travel Assistant Application did not have the `Global Planning` knowledge required to do it.

At last, we want to highlight that if new global planners enter the system, the Travel Assistant Application will be able to know and exploit them in its further execution, thanks to the establishment of new soft dependencies due to the dynamic knowledge extension feature.

5.4 Execution Model Formalization

In this section, we formally define the execution model of an adaptive service-based system defined within our framework. We start by characterizing a deployed instantiation of an adaptive system having running domain object instances. We will then present in details the execution of a domain object, with a focus on the dynamic refinement of abstract activities.

[1] `https://www.fromatob.com/` [2] `http://www.google.com/transit`

The following definition captures the current status of the execution of a given core process. The process instance is a hierarchical structure, obtained through the refinement of abstract activities into fragments. A process instance is hence modeled as a list of tuples process-activity: the first element in the list describes the fragment currently under execution and the current activity; the other tuples describe the hierarchy of ancestor fragments, each one with abstract activities currently under execution. The last element in the list is the process model from which the running instance has been created. A process instance is defined as follows:

Definition 9 (Process Instance)**.** *We define a process instance I_p of a process p as a non-empty list of tuples $I_p = (p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)$, where:*

- *each p_i is a process and $p_n = p$;*
- *$a_i \in A(p_i)$ are activities in the corresponding processes, with $a_i \in A_{abs}(p_i)$ for $i \geq 2$ (i.e., all activities that are refined are abstract).*

An example of process instance is given by the process of the Travel Assistant Application domain object, shown in Figure 5.4, where we reported an example of its execution.

A domain object instance, instead, is specified by its current domain knowledge and the respective instance, and by its process instance, as defined in the following.

Definition 10 (Domain Object Instance)**.** *A domain object instance δ of a domain object $o = \langle \mathbb{DK}_I, \mathbb{DK}_E, p, \mathbb{F} \rangle$ is a tuple $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$ where:*

- *$\mathbb{DK}_{E^+} \supseteq \mathbb{DK}_E$, is the current set of domain properties in the external domain knowledge;*
- *$\bar{l}_I \in \mathbb{L}_{DK_I}$ and $\bar{l}_{E^+} \in \mathbb{L}_{DK_{E^+}}$ are the current state of the domain properties in the internal and external domain knowledge;*
- *I_p is its process instance.*

Notice that $\mathbb{DK}_{E^+} = \mathbb{DK}_E$ when the domain object is instantiated. Then, has highlighted in section 5.3.1, $\mathbb{DK}_{E^+}$ might grow during the domain object execution since, together with fragments to be executed, the

domain object instance receives new domain properties from provider domain objects, thus spanning its original domain knowledge. This mechanism is formally defined later on in the formalization of abstract activity refinement and fragment injection.

We define now an adaptive system instance.

Definition 11 (Adaptive System Instance)**.** *An adaptive system instance AS_I of an adaptive system $AS = \{o_1, \ldots, o_n\}$ is a set of domain object instances $AS_I = \{\delta_{ij}\}$ where each δ_{ij} is an instance of domain object o_i.*

For instance, if we consider the running scenario depicted in Figure 5.4 of the travel assistant system as modeled in Figure 4.3 of chapter 4, we can say that the adaptive system instance, for that specific execution, is made by instances of the `Travel Assistant Application`, `Travel Assistant`, `Journey Planners Manager`, `Journey Manager`, `Data Viewer`, `Rome2Rio`, `Train` and `Bus` domain objects. In other words, the adaptive system is made by instances of the domain objects that our user Sara actually interacted with.

We will now formally define the execution model of domain objects. We start by presenting the refinement of an abstract activity, since it is the core aspect of the proposed approach. In the following a refinement need is formalized.

Definition 12 (Refinement need)**.** *A refinement need is a tuple $\eta = \langle AS_I, \delta, a \rangle$ where:*

- *AS_I is an adaptive system instance;*
- *$\delta \in AS_I$ is the domain object instance for which the refinement is needed;*
- *a is the abstract activity of δ to be refined.*

For instance, considering the process whose refinement is shown in Figure 5.4, the domain object instance for which the refinement is needed is an instance of the Travel Assistant Application, while the abstract activity to be refined is the `Plan Journey` activity.

A refinement is defined as follows.

Definition 13 (Refinement)**.** *A refinement for a refinement need* $\eta = \langle AS_I, \delta, a \rangle$*, denoted with* $REF(\eta)$*, is a tuple* $\langle p_\eta, \mathbb{DK}_\eta, \bar{l}_\eta \rangle$ *where:*

- p_η *is the process to be injected;*
- $\mathbb{DK}_\eta$ *is the set of domain properties to be added to the external domain knowledge;*
- *for each* $a \in A_{abs}(p_\eta)$*,* $Goal(a) \subseteq 2^{\mathbb{L}_{DK_\eta}}$*;*
- $\bar{l}_\eta \in \mathbb{L}_{DK_\eta}$ *is the current state of the domain properties.*

The last two items of the previous definition require that, in case the refinement process contains abstract activities, the domain knowledge needed for their refinement is part of the refinement solution. Indeed, this is how the domain knowledge extension is performed.

We will now characterize a correct solution for a refinement need η. Intuitively, a refinement $\langle p, \mathbb{DK}, \bar{l} \rangle$ is a correct solution to a refinement need $\eta = \langle AS_I, \delta, a \rangle$, if the execution of p brings the external domain knowledge of object δ in a state that satisfies the goal of a. Notice that p, being a composition of fragments provided by other domain objects, might contain abstract activities that will be refined later on, when the refinement is executed. Our definition of correct refinement is based on the assumption that abstract activities, once refined, will behave as declared in their specification (preconditions and effects on their activities). That is, we treat them as all other activities in the process, assuming that their behavior is correctly specified through their annotations in terms of preconditions and effects.

In the following we give the definitions of action executability, action impact, and abstract run of a process. These definitions are the basis for the formal characterization of a correct refinement.

Definition 14 (Action Executability)**.** *An action* a *of a process* p *is executable from domain knowledge state* $\bar{l} \in \mathbb{L}_{DK}$*, denoted with* $Executable(a, \bar{l})$*, if* $\bar{l} \models Pre(a)$ *and the effects of action* a *are applicable in domain knowledge state* $\bar{l}$*.*

In other words, an action is executable from a given domain knowledge state if, in that state, its precondition is verified and its effects can be applied.

Definition 15 (Action Impact)**.** *The impact of action a belonging to some process p when executed from domain knowledge state* $\bar{l} \in \mathbb{L}_{DK}$*, denoted with* $Impact(a, \bar{l})$*, is a domain configuration* $\bar{l'} \in \mathbb{L}_{DK}$ *such that for every* $dp_i = \langle L_i, l_i^0, E_i, T_i \rangle \in \mathbb{DK}$*, if exists an* $e \in Eff(a)$ *such that* $(\bar{l} \downarrow_{dp_i}, e, l_i') \in T_i$ *then* $\bar{l'} \downarrow_{dp_i} = l_i'$*, otherwise* $\bar{l'} \downarrow_{dp_i} = \bar{l} \downarrow_{dp_i}$*.*

The action impact is given by the domain configuration in which the domain knowledge of the domain object that is executing the activity evolves.

Definition 16 (Abstract Process Run)**.** *Given a process* $p = \langle S, S_0, A, T, Ann \rangle$ *and a domain knowledge state* $\bar{l} \in \mathbb{L}_{DK}$*,* $\pi = (s_1, a_1, s_2, \ldots, a_{n-1}, s_n)$ *is an abstract run of p from* $\bar{l}$ *if:*

- $s_1 \in S_0$ *and* $\forall i, \in [1, n] : s_i \in S$*;*
- $\forall i \in [1, n1] : a_i \in A$ *and* $(s_i, a_i, s_{i+1}) \in T$*;*
- *there exists a domain knowledge evolution of* $\mathbb{DK}$*,* $\pi_{DK} = (\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n)$ *such that:*
  - $\bar{l_1} = \bar{l}$*;*
  - $Impact(a_i, \bar{l_i}) = \bar{l}_{i+1}$ *for all* $i \in [1, n-1]$*;*
  - $Executable(a_i, \bar{l_i})$ *for all* $i \in [1, n1]$*.*

A process run that terminates in a state with no outgoing transitions (final state) is called a complete run. We denote with $\Pi_{ABS}(p, \bar{l})$ the set of all possible complete abstract runs of process p from domain knowledge state $\bar{l} \in \mathbb{L}_{DK}$.

We can now define a correct refinement.

Definition 17 (Correct Refinement)**.** *Given a refinement need* $\eta = \langle AS_I, \delta, a \rangle$*, with* $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$*, we say that a refinement* $\langle p_\eta, \mathbb{DK}_\eta, \bar{l}_\eta \rangle$ *is a correct solution for* η*, if for each complete abstract run* $\pi \in \Pi_{ABS}(p_\eta, \bar{l}_{E^+})$*, its associated domain knowledge evolution* $\pi_{DK} = (\bar{l}_1, \bar{l}_2, \ldots, \bar{l}_n)$ *is such that* $\bar{l}_n \models Goal(a)$*.*

Intuitively, a refinement is a correct solution for a refinement need if all its complete abstract runs satisfy the goal of the abstract activity to be refined.

As regards the execution of an adaptive system instance, intuitively, it evolves in three different ways. First, through the execution of activities in domain object instances, which will be presented in detail in the following. Second, through the interaction among domain object instances, which happens according to the standard rules of peer-to-peer process communication. Third, through a change in the behavior, or entrance / exit, of domain objects and domain object instances into the system.

In the following we formalize the execution model of a domain object, considering also the injection of a refinement solution in the case in which an abstract activity is executed.

Definition 18 (Action Execution in a domain object Instance)**.** *Given a domain object instance* $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$*, with* $\delta \in AS_I$ *and* $I_p = (p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)$*, the execution of action* a_1*, denoted with* $exec(\delta, AS_I)$*, evolves* δ *to* $\langle \mathbb{DK}_I, \mathbb{DK}'_{E^+}, \bar{l}'_I, \bar{l}'_{E^+}, I_p \rangle$*, where:*

- *if* $a_1 \in A_{in}(p_1) \cup A_{out}(p_1) \cup A_{con}(p_1)$ *then*
  - $\mathbb{DK}'_{E^+} = \mathbb{DK}_{E^+}$*;*
  - $\bar{l}'_I = Impact(a, \bar{l}_I)$ *and* $\bar{l}'_{E^+} = Impact(a, \bar{l}_{E^+})$*;*
  - *if* $next(p_1, a_1) \neq \emptyset$ *then* $I'_p = (p_1, next(p_1, a_1)), (p_2, a_2), \ldots, (p_n, a_n)$*, otherwise* $I'_p = (p_2, next(p_2, a_2)), \ldots, (p_n, a_n)$*.*
- *if* $a_1 \in A_{abs}(p_1)$*, given* $\langle p_\eta, \mathbb{DK}_\eta, \bar{l}_\eta \rangle = REF(\eta)$*, with* $\eta = \langle AS_I, \delta \rangle$*, then*
  - $\mathbb{DK}'_{E^+} = \mathbb{DK}_{E^+} \cup \mathbb{DK}_\eta$*;*
  - $\bar{l}'_{E^+} \in \mathbb{L}_{DK'_{E^+}}$ *is such that for every* $dp_i = \langle L_i, l_i^0, E_i, T_i \rangle \in \mathbb{DK}'_{E^+}$*, if* $dp_i \in \mathbb{DK}_\eta$ *then* $\bar{l}'_{E^+} \downarrow_{dp_i} = \bar{l}_\eta \downarrow_{dp_i}$*, otherwise* $\bar{l}'_{E^+} \downarrow_{dp_i} = \bar{l}_{E^+} \downarrow_{dp_i}$*;*
  - $I'_p = (p_\eta, a_\eta^0)(p_1, a_1), (p_2, a_2), \ldots, (p_n, a_n)$*.*

Eventually, we previously said as a soft dependency among two domain objects becomes a strong dependency, denoted with $\delta_{ih} \leftarrow \delta_{jk}$, if the domain object δ_{ih} injects in its internal process a fragment provided by δ_{jk}. This is formally defined as follows:

Definition 19 (Strong Domain Objects' Dependency)**.** $\forall \delta_{ih}, \delta_{jk} \in AS_I$ *with* $i \neq j$ *and* $h \neq k$, $\delta_{ih} \leftarrow \delta_{jk}$ *if* $\exists (f, a) \in I_p(\delta_{ih}) | f \in \mathbb{F}(o_j)$.

In the next section, we show how the refinement problem previously presented can be solved by applying the automated fragment composition approach based on AI planning [25, 120].

5.4.1 Automated Refinement via AI Planning

Within the approach presented in [47] and summarized in section 5.1, we said that a fragment composition problem is transformed into a planning problem. Relevantly to our purposes, such techniques cover uncertainty, in order to allow the composition of services whose dynamics is only partially exposed, and is able to deal with complex goals and data flow [120].

In the following we briefly describe how a refinement need $\eta = \langle AS_I, \delta, a \rangle$, with $\delta = \langle \mathbb{DK}_I, \mathbb{DK}_{E^+}, \bar{l}_I, \bar{l}_{E^+}, I_p \rangle$ is transformed into an AI planning problem. In other words, we say how the approach in [47] is adjusted and used in our framework.

First of all, a set of n fragments, $(f_1, \ldots, f_n)$, is selected from the soft dependencies of δ: for some $\delta' \in AS_I$, with $\delta \dashleftarrow \delta'$, $f_i \in \mathbb{F}(\delta')$.

Advanced optimization techniques, as the one described in [128], can be used to further reduce the set of fragments on the basis of the functionalities they provide and of the preconditions satisfiability of their preconditions in current domain knowledge state. Both fragments $(f_1, \ldots, f_n)$ and the set of domain properties $(dp_1, \ldots, dp_m) \in \mathbb{DK}_E^+$, on which the fragments are annotated, are transformed into state transition systems (STS) using transformation rules similar to those presented in [47]. During this encoding, all goals on abstract activities in fragments are ignored, while preconditions and effects are maintained. With this measure, the refinement plan will be built under the assumption that abstract activities will behave according to their annotation, independently from the way in which they will be refined (see Definition 17).

The planning domain Σ is obtained as the product of the STSs $\Sigma_{f_1} \ldots \Sigma_{f_n}$ and $\Sigma_{dp_1} \ldots \Sigma_{dp_m}$, where STSs of fragments and domain properties are synchronized on preconditions and effects, $\Sigma = \Sigma_{f_1} \| \ldots \| \Sigma_{f_n} \| \Sigma_{dp_1} \| \ldots \| \Sigma_{dp_m}$. The initial state of the planning domain is derived from the

initial state of all fragments and the current state of the domain properties $\bar{l}_{E^+}$, by interpreting it as states of the STSs defining the planning domain. Similarly, the refinement goal $Goal(a)$ is transformed into a planning goal ρ by interpreting the states in $\mathbb{DK}_{E^+}$ as states in the planning domain.

Finally, the approach of [120, 47] is applied to domain Σ and planning goal ρ to generate a plan Σ_η that guarantees achieving goal ρ once "executed" on system Σ. State transition system Σ_η can be further translated into an executable process p_η, which implements the identified solution.

5.5 Discussion

In this chapter, we described the *adaptation mechanisms and strategies* exploited by our design for adaptation framework. In particular, these realize the dynamic adaptation of fragment-based and context-aware processes proposed in [25, 120] and based on AI planning [47]. Moreover, the mechanisms that we use implement a *goal-based* approach. Indeed, also our opinion is that goal-based approaches are more robust against typical changes in dynamically evolving environments, compared to rule-based and built-in methodologies. As also emerged from the motivating example of this dissertation, indeed, the dynamicity of the environment is not only given by service availability or changes in performance values of services, but it is especially due to the continuous emergence of new services, to the evolution of both the domain and the context in which service-based systems operate, to changes in the implementation of services, as well as to changes in business policies that can affect their operation, etc.

In this context, goal-based approaches are more *flexible* and *robust*, and they can benefit from the availability of automated service composition techniques dealing with abstract composition requirements. For these reasons, we selected the approach in [25, 120] for the execution and adaptation of service-based applications realized within our framework, and we defined a design for adaptation approach allowing to support and facilitate its application.

In order to comprehensively understand how these mechanisms and strategies have been embedded in our framework, we described the en-

ablers of the design for adaptation framework, comprising the *process engine* and the *adaptation engine*, and we detailed how they communicate and inter-operate to perform, respectively, the *execution* and the *adaptation* of systems.

In addition, through a running scenario we have shown how service-based systems perform at run-time and we also highlighted important features of our approach, making it further suitable in dynamic environments, namely *higher order abstract activities* and the *dynamic knowledge extension*. The first feature allows for the run-time definition of goals labeling abstract activities in processes and fragments, when they strongly depends on the run-time execution context (e.g., by the user choice) in order to define them when specific data and required services are known. The second feature, instead, refers to the possibility for services, modeled as domain objects, to dynamically span their view on the operational environment, while executing. This happens thanks to the application of the abstract activities refinement mechanism that enables a *higher level of dynamicity*, allowing domain objects to dynamically discover other available domain objects providing their required functionalities on the need.

A current limitation of the presented approach, as discussed in this chapter, is that while the adaptation engine currently implements all the adaptation mechanisms and strategies reported in section 5.1, in our design for adaptation framework, at the moment, we only handle the refinement mechanism. We plan, as future work, to extend the framework to the management of all the other mechanisms. However, we specify that this will ask more for an implementation effort that for an extension of the domain objects model, which already includes all the required constructs to handle the local and compensation adaptation mechanisms (e.g., preconditions, effects, goals).

The following chapter is devoted to an extension of the design for adaptation approach presented in this dissertation, to make it suitable for the modeling and execution of *Collective Adaptive Systems* (CAS).

Chapter 6

Domain Objects for Collective Adaptive Systems

In previous chapters, and through the description of the travel assistant system, we discussed how modern service-based systems are progressively becoming more heterogeneous. They actually form socio-technical systems, composed of distributed entities (e.g., services and service providers, software and human participants) interacting with and within the environment. Moreover, we have seen how these systems operate under constant perturbations that are due to unexpected changes in the environment and to the changes in the behavior of the participants (i.e., services, users).

In previous chapters, we have seen how adaptive service-based systems can be modeled, through domain objects (in chapter 4), and how they operate in open and dynamic environments, through incremental and dynamic adaptation (in chapter 5), from a *single-user* point of view. However, the level of complexity of modern systems comes specifically from bringing together and combining in the same operating environment heterogeneous and autonomous components, systems and users, with their specific concerns. In this context, multiple participants must adapt their behavior in concert to respond to critical run-time impediments. Furthermore, also by considering the mobility domain as an example, we already introduced the collective nature of some mobility services (e.g., ride-sharing), as those belonging to the emergent shared economy trend. As a consequence, we argue that for a service-based system to be resilient, adaptation must be *collective*.

With these premises, in this chapter we deal with the design and execution of Collective Adaptive Systems. The term *Collective Adaptive Systems* (CAS from here on) has been introduced in the literature to denote large-scale systems that may present substantial socio-technical embedding [129]. They typify systems with complex design, engineering and management, whose level of complexity comes specifically from bringing together and combining in the same operating environment heterogeneous and autonomous entities, systems and users, with their specific concerns. We have already seen that to be robust against the high degree of dynamism of their operating environments, and to sustain the continuous variations induced by their socio-technical nature, these systems need to *self-adapt*. Indeed, self-adaptation is an important feature of complex software systems. However, it is often seen as a means to automate management activities in order to meet desired requirements, such as minimizing resources and costs (e.g.[130]). In a CAS instead, self-adaptation is a feature of the collectiveness. Individual entities may "opportunistically" enter the system and self-adapt in order to leverage other entities' resources, functionalities and capabilities to perform their task more efficiently or effectively. But, the collaborative nature of the system, makes this self-adaptation much trickier [131]. Changes in the behavior of one entity may break the consistency of the whole collaboration, or have negative repercussions on other participants.

Adaptation must, therefore, be *collective*. Entities must be able to self-adapt simultaneously and, at the same time, preserve the collaboration and benefits of the system (or sub-system) they are within. Self-adaptation of an individual entity is therefore not only finalized to the achievement of its own respective goals but also to the fulfillment of emerging goals of the dynamically formed sub-systems.

To model CAS within our framework, extensions to the domain objects model are required, in order to deal with both the collective nature of services and their collective adaptation. In this chapter, we introduce and detail (1) the extensions made to our approach to make it suitable for modeling CAS; (2) the *collective adaptation algorithm* defined to manage the collaborations among entities allowing them to simultaneously self-

adapt, possibly without breaking the consistency of the collectiveness.

This chapter is mainly founded on the work presented in [30] and it further extends it. In particular, the work in [30] has been performed in the context of the ALLOW Ensembles European project [35] that is part of the Fundamentals of Collective Adaptive Systems (FoCAS) initiative [1].

The rest of the chapter is structured as follow. We start by giving an overview on CAS in section 6.1. To demonstrate the generality of the approach, we apply it to two application domains: the *Urban Mobility* and the *Surveillance* systems, which are described in section 6.2. Section 6.3 is devoted to the extensions made to the domain objects model to design CAS (in section 6.3.1), and to the definition of the *collective adaptation algorithm* to perform collective adaptation (in section 6.3.2). The formal model of the extended approach is given in section 6.4 while, its evaluation applied to the two different applications domains is reported in section 6.5. In conclusion, a discussion closes the chapter in section 6.6.

6.1 Related Work on Collective Adaptive Systems

In this section we will review recent works on *coalition formation* and *decision-making for multi-agent systems*, on *component ensembles* and on *run-time adaptation in multi-agents systems.*

Coalition formation has been widely studied in game theory and economics. In multi-agents system, and more in general in Artificial Intelligence (AI), coalition formation has been used as a means for dynamically creating partnerships or teams of cooperating agents. The problem involved in the maintenance of cooperation in a social group is generally called the "social dilemma", for which the n-person Prisoners' Dilemma game is often used as a typical toy example.

According to [132], if we view the population of agents A as a set, then each subset of A is a potential coalition. Coalitions in general are goal-directed and short-lived; they are formed with a purpose in mind and dissolve when that need no longer exists. Within a coalition, the organizational structure is typically flat, although there may be a distinguished

[1] `http://www.focas.eu/`

"leading agent" which acts as a representative and intermediary for the group as a whole.

Many works on coalition formation in multi-agents systems, e.g. [133, 134, 135], use the assumption that all agents (or a subset) can directly communicate with each other, which is not realistic in the real world. [136] tackles the problem of coalition formation in multi-agents systems in a neighbourhood agent network (a network in which agents communicate directly only with their neighbours). Agents can participate to several coalition at the same time, by indicating for each of them the *degree of involvement* (DoI). A coalition is initiated by an agent issuing a task; such an agent is called initiator. Then the initiator contacts its neighbours trying to find a subset of them satisfying the proposed task. If there is no a subset of its neighbours able to satisfy the task, then the initiator randomly selects a neighbour as mediator, in order to contact the mediator neighbours.

A task is satisfied when the coalition that has been formed have the resources required by it. When the initiator contacts its neighbours, then a negotiation takes place to determine what is the degree of involvement of the agent. The negotiation protocol allows an agent to make multiple agreements with other agents and to cancel temporary agreements without paying penalty. Indeed during the coalition formation all the agreements made by the agents with the initiator are considered temporary and can be aborted by both the initiator and the agents before the coalition formation. When a coalition is created, the agreements cannot be cancelled. Negotiation is based on offer/counteroffer accept protocol. During its life in a coalition, an agent can adapt its DoI in order for example to join another coalition. By modifying its DoI in a particular coalition, an agent will pay a penalty. Hence adaptation mechanism is in charge of adjusting agent DoI values in current coalitions to minimize the agent penalty when it joins a new coalition.

Multi agents systems have been applied in a variety of applications ranging from supply chains management [137], to complex software systems building [138], and to multi-satellite mission planning [135]. Here, we take in consideration solutions proposed in different areas.

Regarding the considered ***agent type***, there exist few solutions which

are application specific i.e. we regard them as homogeneous agents (e.g., [139], [136]) while most of the related work is more general and it includes agents with different types of roles i.e. heterogeneous agents (e.g., [140], [141], [142], [135]).

Regarding the ***agent behavior*** (i.e., whether the agent acts in a selfish or cooperative way), while some of the works include solutions where the agents cooperate to achieve a specific common goal, typically the literature in game theory assumes utility maximizing agents. They cooperate if they have the right incentives. They break from a coalition if they can do better with another coalition, or alone.

Regarding the ***coalition management strategy*** exploited (i.e., Decision Making) we have few studies where in the solution a centralized coordinator is designated to collect information from all the agents and then to disseminate a decision to the whole group. However, such a strategy interferes with the system's scalability and robustness: the coordinator can easily become a communication bottleneck, and it is also a potential point of failure for the system. Because of that, most of the works mentioned here propose decentralized approach. As remarked in [143], the decision making in large distributed systems is a difficult problem.

The *centralized decision making* approach (which is the one typically adopted) is to collect all the information from the agents in a single center and solve the resulting optimization problem (see, e.g., [144]). Such centralized solution for a sufficiently large number of agents becomes impractical [143]. However, these difficulties can be overcome by allowing the agents to cooperate or self-organize in solving this distributed decision problem. The main issue in this decentralized approach is identifying which agents need to communicate, what information should be sent, and how frequently [143]. The work in [143], deal with the multiagent collaboration problem by using analytical techniques. In particular, it requires that several assumptions be made about the form of the problem: (i) the decisions of the individual agents are represented by elements of a continuous and finite-dimensional vector space; (ii) the agents are coupled via a shared objective function that is continuous and twice differentiable, and (iii) there are no interagent constraints.

Biological collective behaviors, where coordinated global behavior emerges

from local interactions, have inspired decentralized approaches in a large variety of application domains, including mobile robot exploration, sensor network clock synchronization, and shape formation in modular robots [145]. Research effort has also been devoted to addressing coordination challenges in large multi-agent systems where agents are spatially distributed or heterogeneous. For instance, it has been provided support for reaching group consensus by one or a few better-informed agents (see, e.g., the work in [145]), where it is shown how the number of informed agents (i.e., the ones able to obtain important information not available to others) and their confidence levels affects the consensus process.

Master-slave schema has also been adopted for context-driven dynamic agent organization (e.g., in the MACODO organization model [146]). In such type of agent organisation, the master agent has complete knowledge of the organisation state and controls the organisation dynamics in a centralised way. Master agents of different organisations can cooperate (exchanging among each other some reduced state information) in order to achieve some common goal (for example, by merging their respective set of agents into a single organisation). Finally, in [139] is presented a decentralized and dynamic method where coalition formation is achieved by opportunistic aggregation of agents, while maximizing coalition benefits by means of taking advantage of local resources in the grid.

In the literature there exist lot of solutions regarding a coalition of coordinating agents where each agent plays a specific role known as *choreography*. More in general, a service choreography is a distributed service composition in which services interact without a centralized control. Several work have faced the problem of adaptation in service choreography, see [147] for an exhaustive survey. For instance, [140] proposes a framework for programming distributed adaptive applications. Applications are programmed using a choreographic language (called AIOC) suited for expressing patterns of interaction from a global point of view. AIOC allows the programmer to specify which parts of the application can be adapted. Adaptation takes place at runtime by means of rules, which can change during the execution to tackle possibly unforeseen adaptation needs. The framework is also endowed with formal proofs of correctness and deadlock-freedom.

A group of interacting agents can also be seen as an *ensemble*. SCEL [141] is a formal language that provides abstractions for autonomic systems in terms of behaviors, knowledge, aggregation and policies. The most interesting part for us it the abstraction for aggregation. In SCEL it is possible to define ensembles as a set of components, and the choice of which components participate at runtime to a certain ensemble is based on the satisfaction of certain predicates. Moreover SCEL exploits predicate-based communication, in order to support ensemble-based communication.

[148] presents an ensemble based component model, based on the idea that the only way for components to bind and communicate is via an ensemble. Hence an ensemble embodies a dynamic binding among a set of components and thus determines their composition and interaction. As in [141] the membership to a particular ensemble is determined by predicates. Moreover, components belonging to an ensemble are not able to direct communicate with each other, there has to be an ensemble coordinator that is in charge of sharing the knowledge among all the participants.

[142] gives a formal foundation for ensemble modelling. An ensemble is defined in terms of roles and role connectors. A role can be seen as a meta-component (or a type) that can be instantiated by components of different types. Role connectors specify the I/O operations among roles. A role is endowed with a cardinality, thus it is possible to control at runtime the number of component playing a particular role. Moreover the runtime behaviour of an ensemble is given by means of an automaton. Noteworthy, a component participating into an ensemble can adopt a role (either by entering into an ensemble or by playing an additional role) or can drop a role. The idea of role is close to ours, but in our approach instead of defining connections among roles in the ensemble, we specify consistency rules for the adaptation, by leaving to cell the freedom of interaction.

Concerning the ***hierarchical structure of coalition***, allowing approaches to deal with systems of systems, there exist only few studies where the approaches take in consideration the hierarchy of the system, thus contributing to scalability. Example of a coalition formation mechanism which allows a hierarchical structure is presented in [142]. The approach is named *Helena - Handling massively distributed systems with ELaborate ENsemble Architectures* and it represents a modelling technique centred around

the notion of roles teaming up in ensembles. Ensembles are built on top of a component-based platform as goal-oriented communication groups of components. The functionality of each group is described in terms of roles which a component may dynamically adopt.

Concerning the ***decentralized self-adaptation***, we have some studies which discuss about approaches which can deal with adaptation. In order to deal with increasingly growing complexity of the mission-critical software systems, there are some recent works which take into account adaptation at run-time. Contemporary mission-critical software systems are often expected to safely adapt to changes in their execution environment where run-time adaptation mechanisms reduce the complexity of the system. For example, in [149] is presented an adaptive run-time model used to establish a flexible information processing within a group of heterogeneous robots, while in [150] is presented a reusable framework for developing adaptive multi-robotic systems for heterogeneous robot teams using an organization based approach.

Existing works for distributed self-adaptive systems are typically based on multi-agent and MAPE loop paradigms [151], [152]. More specifically, the system is decomposed in self-handling software units, which collaborate and coordinate in a distributed way. In [135], for example, the life-cycle of each agent is decomposed into three steps: "Perceive - Decide - Act" where the "Decide" phase is the key step where the agent chooses with action it has to perform using its partial perceptions.

In [153], it is presented a rigorous approach to decentralising the control loops of distributed self-adaptive software used in mission-critical applications. Specifically, it uses quantitative verification at runtime, first to agree individual component contributions to meeting system level quality-of-service requirements, and then to ensure that components achieve their agreed contributions in the presence of changes and failures. All verification operations are carried out locally, using component-level models, and communication between components is infrequent.

Discussion.

In conclusion, we reviewed these works according to some criteria, such as (i) the *agents type* (i.e., homogeneous or heterogeneous); (ii) the *agents*

behavior (i.e., selfish or cooperative); (iii) the *coalition management strategy* (i.e., centralized or distributed); (iv) the *hierarchical structure of coalitions* (to support systems of systems); and, (v) the *adaptation application* (i.e., design-time or run-time adaptation). What emerges from our review is essentially that each work focuses only on one or few criterion. To the contrary, there is the need for approaches dealing concurrently with different aspects, and possibly all the before-mentioned features, making them able to manage complex systems, such as CAS. These approaches should be able to manage both cooperative and selfish behavior between agents.

Furthermore, it is extremely important that the management of coalition is decentralized, in such a way of eliminating the single point of failure and potential bottlenecks in the system.

Lastly, the adaptation must be performed at run-time to deal with the openness and dynamicity of the environment.

This chapter addresses the challenge of *collective adaptation in service-based systems*, by exploiting the *design for adaptation* approach, presented in this dissertation, that leverages also on the key features of service-oriented design to support the modeling, development, and execution of CAS operating in dynamic environments. Key properties of our approach are the emphasis on collaboration towards fulfillment of individual diverse goals and the heterogeneous nature of the system with respect to roles, behaviors and goals of its participants. These properties distinguish our approach from other types of collective adaptation approaches, like for instance swarms, where all elements of a community have a uniform behavior and global shared goal [154], and multi-agent systems and agent-based organizations [155], where there may be several distinct roles and behaviors, but the differentiation is still limited and often pre-designed. While existing approaches normally deal with CAS through isolated adaptation, we propose a framework to build CAS that fully addresses the challenge of collective behavior of systems by fulfilling the following requirements:

R1. Support for a highly dynamic environment and an open and dynamic nature of the system where entities communication and adaptation handling are context-aware.

R2. Support for large-scale distributed configuration of the system, with different decision management strategies (from *hierarchical* to *peer–to–peer*).

R3. A collective adaptation approach allowing entities to collectively adapt at runtime and in a decentralized manner, guaranteeing the reliability of the system.

The approach is evaluated in two different application domains, namely a *Urban Mobility System* (see section 6.2.1), aiming at the management of multi-modal and *collaborative* mobility in a smart city, and a *Surveillance System* (see section 6.2.2), devoted to the detection and handling of intruders in private companies. The application and evaluation of the approach in these very different domains allows us to demonstrate its *scalability* and *reliability* when applied to real-world scenarios, as well as to prove its capability of successfully deal with different problems, both in terms of scope and nature. In the next section, we describe the two scenarios that will drive us through the chapter.

6.2 Application Scenario Examples

To better understand the class of systems we intend to approach, in this section we give two motivating examples of systems characterized by the aforementioned requirements. In particular, these systems are both characterized by the *collective* and *cooperative* nature of their services.

6.2.1 The Urban Mobility System

This example refers to the specification of a multi-modal and collaborative Urban Mobility System (UMS). Differently from the travel assistant presented in section 3.1 that was focused on single users, the goal of the UMS consists in the collaborative exploitation of the city transport facilities, while providing real-time, and customized mobility services for the whole travel duration. To this aim, the UMS exploits a variety of *heterogeneous* services, from city mobility resources (e.g., traditional public transportation, bike sharing, flexibus), with their transport service functionalities

(e.g., registration, booking), to general-purpose ones (e.g., different payment services). All these services are often provided by *autonomous* entities. The UMS purpose is twofold: (i) integrate the available services sup-

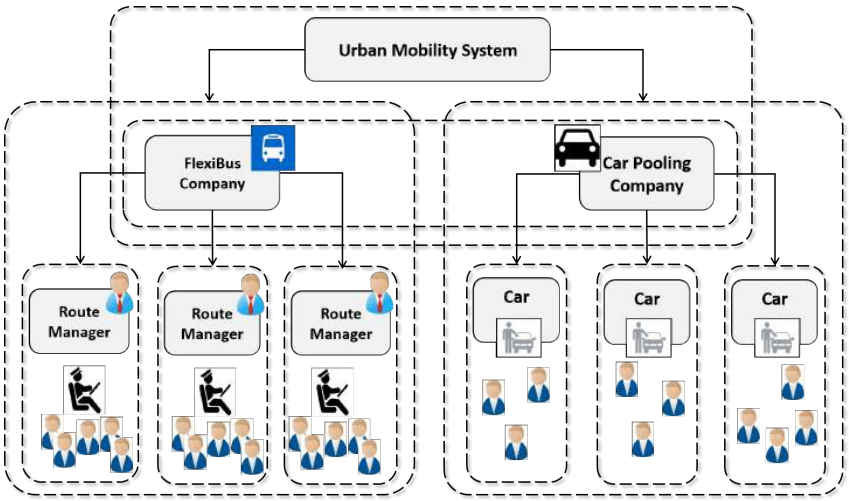


Figure 6.1: Urban Mobility System: an overview.

porting the planning (multi-modal journey planners), organization (booking and payment services), and execution (real-time availability of services) of a trip; (ii) help citizens to deal with context changes that may affect a journey, with the introduction of alternatives or different planning solutions, by exploiting the adaptation abilities of the involved entities, in a collective manner. Moreover, the UMS operates in a continuously changing and complex environment. Each service may enter or leave the system at any time (e.g., a new bike-sharing service), as well as it may change or extend its offered functionalities, making the system *open* and *dynamic*. The dynamism is also given by the systems context, whose change can affect the operation of the system (e.g., traffic jams, bus delays, on-line payment services unavailable) (**R1**). The UMS is characterized by the involvement of different entities playing different roles. First of all, it requires an active involvement of citizens. Then, each transport service is handled by an entity manager, acting also as a mediator between the passengers and third party services. Drivers are also involved, by dynamically interacting both with the system and with the passengers of their routes. Finally, the UMS supervises the whole system, by providing integration between all the involved parts. The system results in an adaptive *hierarchy* of entities, as

shown in Figure 6.1, that dynamically evolves in response to the evolution of the environment and the involved entities (**R2**). Moreover, in order to support a sustainable mobility within the urban environment, the system's intent is that of inducing users to prefer collective mobility solutions (e.g., flexibuses, car sharing), both during the normal execution of the system and in case an adaptation need arises (e.g., intense traffic on the route). To this aim, each entity is able to interact with other entities, to *notify*, *solve* or *manage* problems, as well as to adapt its own process in order to apply collective and adaptive solutions (e.g., the flexibus changes its path) (**R3**). In particular, in the scope of our scenario, we focus on the flexibus and car pooling collective mobility services. Flexibus is a modern transportation service, that combines the features of taxi and regular bus services. A flexibus system manages on-demand routes defined as a network of stops (pickup points) and provides to passengers transportation between any of these points. Car pooling, instead, provides integration between independent drivers and passengers.

6.2.2 The Surveillance System

This example refers to a Surveillance System (SurSys) for the premises of a private company [156]. The objective of the system is the detection of intruders entering the factory buildings. The system involves *heterogeneous* entities: (i) *smart devices*, such as *Unmanned Aerial Vehicles* (UAVs) following specific protocols as defined in the service agreement of the company; *movable cameras* placed on the top of the buildings; *fixed sensors* in strategic places monitoring movements in the environment. Next there are (ii) *physical persons*, such as *guards* controlling the surveillance process and *maintainers* that are in charge of maintaining the used equipment (e.g., drones, cameras, or sensors). Eventually, (iii) *manager entities* for the whole management of the system, such as *ground stations* handling multiple drones and receiving telemetry data, also used to do simple recalculations of the missions, and a *central station* that can be located somewhere in the cloud. As regards the behavior of the SurSys, besides the application logic devoted to the detection of intruders, from an adaptation point of view, each entity provides its adaptation logic to be played in case

adaptation needs arise (e.g., the detection of obstacles by a drone).

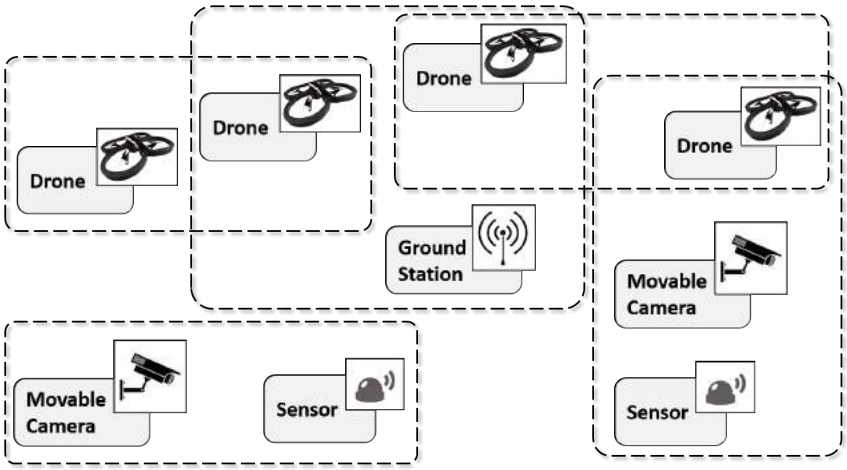


Figure 6.2: Surveillance System: an overview.

The number and the type (e.g., new type of drone/sensor) of the entities in the system can vary according to the size of the area, the number of buildings and other characteristics of the company that wants to use the SurSys. Moreover, services can change their functionalities, or offer new ones (e.g., a drone starting to take photos), requiring for an open and dynamic system (**R1**). Differently from the UMS, the SurSys follows a decision management strategy that is a mix between the *peer–to–peer* and the *hierarchical*one that better fits its characteristics. Indeed, in this system there is neither a central supervisor nor different abstraction levels between the involved entities. In contrast, according to the current context and to the specific goal to reach, entities can dynamically organize themselves in sub-systems which collaborate in a peer–to–peer manner to accomplish their work (see Figure 6.2) (**R2**). Moreover, the entities must avoid to bring the system to a halt, being able to react to dynamic context changes. To this aim, entities must collaborate in order to better deal with unexpected problems, by adapting their own processes applying collective solutions (e.g., the Maintainer can manually drive a drone with problems, while the Ground Station sends a new drone) (**R3**).

In the next section, we present the extensions made to the design for adaptation models (i.e., domain objects model) to make it suitable for the management of CAS.

6.3 Domain Objects for Collective Adaptive Systems

In this section, we present the extensions applied to our design for adaptation approach in order to modeling CAS (Section 6.3.1), and the collective adaptation algorithm that has been realized to perform the collaboration among the entities of a CAS, in cases where it needs to adapt (Section 6.3.2). Both the design approach and the collective adaptation algorithm have been defined by addressing the three requirements identified in section 6.1.

6.3.1 Modeling of Collective Adaptive Systems

The Domain Object model presented in chapter 4 has been defined with the purpose of modeling and executing adaptive service-based systems.

To allow system entities to collectively adapt, dealing with adaptation needs that can be raised both by the environment and by the entities themselves, the model has been extended with specific constructs. Indeed, while till now we used domain objects to model services, we aim now to exploit them to model entities. Moreover, in CAS the concept of role is also relevant. A role models the way in which different entities can interact and collaborate (e.g., bus driver, bus passenger, bus company). A key concept are *ensembles*, that are modeled over the dynamic network of domain objects, as groups of domain objects and that, although autonomous in their execution, share common goals and might need to collectively handle runtime adaptation problems. For instance, in the UMS scenario, an ensemble is that made by the *flexibus driver*, the *route manager* handling the specific route, and all the *passengers* subscribed to the route, as highlighted in Figure 6.1 by dotted lines. In our framework, ensembles are modeled by using an XML-like language, as reported in Figure 6.3, where an example of the before-mentioned ensemble is given. Each participant is modeled as a domain object and behaves autonomously, but if something occurs (e.g., the route is blocked), they should collectively adapt to fulfill collective goals (e.g., being on time at the destination point). Moreover, ensembles can also involve entities at different levels (as highlighted in Figure 6.1 with dotted lines). An ensemble can be made by the *flexibus company* and the

car pooling company, to allow different companies to support each other, in case of adaptation needs spanning over the scope of a single transportation mean. Lastly, when an *intra-ensemble* adaptation can not be solved, *inter-ensembles* adaptation can be performed.

```
<tns:ensemble name="RouteA" xmlns:tns="http://das.fbk.eu/Ensemble"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://das.fbk.eu/Ensemble.xsd">
    <tns:domainObjectType>RouteManager</tns:domainObjectType>
    <tns:domainObjectType>FlexibusDriver</tns:domainObjectType>
    <tns:domainObjectType>RoutePassenger</tns:domainObjectType>
</tns:ensemble>
```

Figure 6.3: Example of the Route Ensemble Model.

In Figure 6.4 we depict a domain object (as the one reported in Figure 4.2) equipped with the new constructs extending it. In the following, we

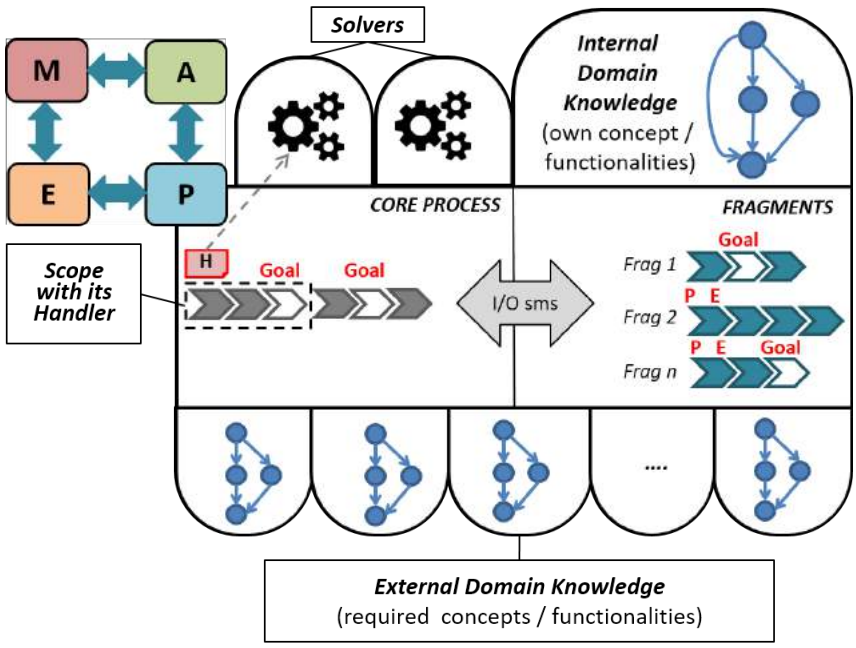


Figure 6.4: Extended Domain Object.

describe these new elements and their usage.

To enable collective adaptation, each domain object implements a set of *collective adaptation solvers* (from here on, simply *solvers*), as well as

a set of *collective adaptation handlers* (from here on, simply *handlers*) (see Figure 6.4).

Solvers model the ability of a domain object to handle one or more *issues*. Each solver relates to the particular issue that it can handle. Since the environment changes frequently and unpredictably, the system requires constant monitoring. Handlers are used to capture issues, during the nominal execution of a domain object, and to trigger the appropriate solver. Each handler refers to a finite *scope* in the core process of a domain object. An example of scope is shown in Figure 6.5 and it refers to the core process of the domain object implementing the role of a Flexibus Driver. Furthermore, an handler can be of two different types:

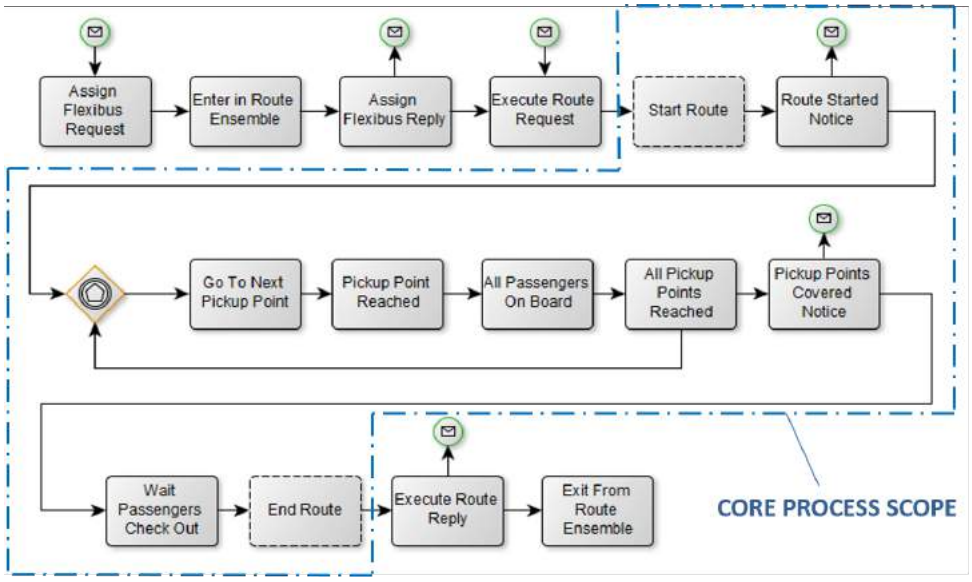


Figure 6.5: Scope in the Flexibus Driver core process.

- *onExternalIssue* handlers are used to catch issues coming from other domain objects in the system (both in the same or in a different ensembles);
- *onInternalIssue* handlers are devoted to monitor properties expressed on the own knowledge of the domain object, and catch the issues arising when this properties are violated.

For instance, in Figure 6.6 we report the XML listing modeling the handlers related to the scope in the flexibus driver process of Figure 6.5.

The flexibus driver can catch the `Intense Traffic` external issue (lines **7-10**), coming from the external environment, as well as the `Bus Broken` internal issue (lines **11-14**), thanks to its knowledge.

```
1<tns:scope name="FD_scope1">
2    <tns:invoke name="FD_RouteStartedNotice"></tns:invoke>
3        ...
4        <!--Here the set of the activities in the scope to which
5        the following handlers refer-->
6    <tns:eventHandlers>
7        <tns:eventHandler name="IntenseTraffic">
8            <tns:onExternalIssue onEventName="IntenseTraffic"/>
9            <tns:callSolver solverType="Route-IntenseTraffic"/>
10       </tns:eventHandler>
11       <tns:eventHandler name="BusBroken">
12           <tns:onInternalIssue onInternalKnowledge="DP-BusStatus.broken"/>
13           <tns:callSolver solverType="Route-BusDamaged"/>
14       </tns:eventHandler>
15   </tns:eventHandlers>
16</tns:scope>
```

Figure 6.6: Collective Adaptive Handlers Example.

A collective adaptation process is triggered by a run-time occurrence of an extraordinary circumstance corresponding to an issue. The resolution of the issue is the outcome of the triggered adaptation process in which all the affected domain objects adapt collaboratively and with a possible minimal impact on their execution. In the next section, we describe in details the collective, yet decentralized, handling of a collective adaptation.

6.3.2 Execution of Collective Adaptive Systems

During the normal execution of the system, through the interactions running between domain objects, ensembles are formed. In Figure 6.7, we show an example of the Route Ensemble, as modeled in Figure 6.3. In particular, we show how a user interacting with the UMS (e.g., by booking a flexibus ticket), can enter a particular ensemble (e.g., Route Ensemble A) with a particular role (e.g., RoutePassenger). Ensembles can be created spontaneously and change over time: different entities may join or leave an existing ensemble dynamically and autonomously. The termination of an ensemble is also spontaneous. It may occur because the participants have reached their goals, or because the ensemble itself has, at some point, ceased to provide benefits. For instance, during the execution of the UMS,

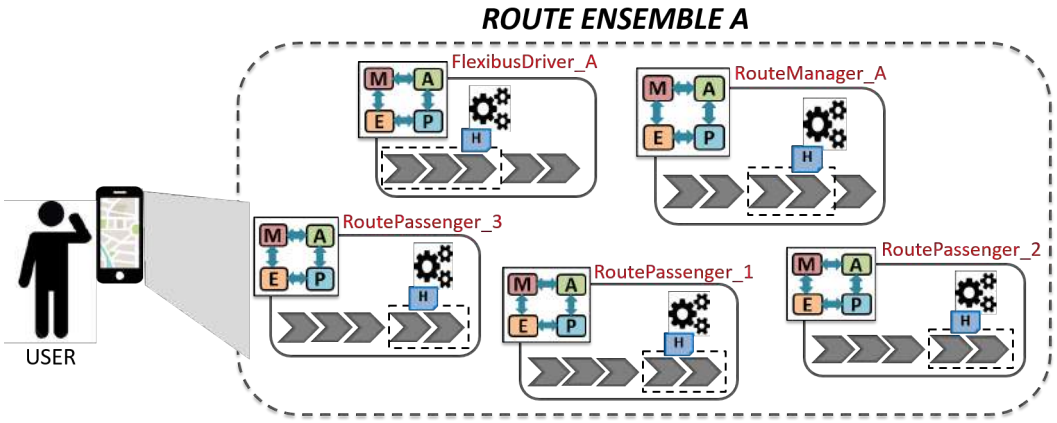


Figure 6.7: Route Ensemble example.

users start to subscribe to a specific flexibus route, by exploiting functionalities of the route manager. This means that the route manager has previously set the route and assigned a driver to it. In this way, the ensemble made by the route manager, the flexibus driver and the passengers is set up.

While the execution goes on, the ensemble can evolve. New passengers can subscribe to the route, while others can leave it.

However, to deal with unpredictable changes, local adaptation is not enough, since the scope of these changes goes beyond the single entity.

Typical changes occurring in dynamic environments are characterized by the fact of affecting different entities:

1. the entity directly related to the change (e.g., a route interrupted directly affects the flexibus driver);

2. the entities with which it interacts, that is the ones belonging to the same ensemble (e.g., both the passengers on board and the ones waiting at the bus stops);

3. the entities that are involved as a consequence of the adaptation executed to solve the problem (e.g., the UMS is consulted to find a new plan for the waiting passengers); these entities can belong to different ensembles with respect to the entity raising the initial issue.

This demonstrates the need for collective adaptation approaches able to deal with dynamic changes, and whose scope can be, in the worst case, the entire system. Thus, such an approach must provide one or more decision management strategies, in order to allow different entities to communicate and cooperate in a collective manner.

> The collective adaptation process is handled in a decentralized manner by the domain objects involved, directly or indirectly, in an adaptation need.

As depicted in Figure 6.4, each domain object implements a *Monitor - Analyze - Plan - Execute* (MAPE) loop [157] that allows for the dynamic interaction with the other domain objects. In Figure 6.8, the *state machine* (SM) representing the operation of a MAPE loop is shown. For readability reasons, we use a color code, which is assigned to the four phases of a MAPE loop. In the following, we highlight the most interesting states of the SM. In the *Monitoring* phase, each domain object executes its core process, while monitoring the environment through active handlers. Issues can come both from the domain object itself (`Issue Triggered` state) or from a different domain object, which asks support for solving an issue (`Issue Received` state). This starts the *Analysis* phase, where the issue solver is called (`Local Solver Called` state). In the *Planning* phase, if the solver has found a solution (`Solution Found` state), the *collective planning* phase starts (gray area in Figure 6.8). All the domain objects involved in the issue resolution process will collectively collaborate to solve the issue. Here, we explore the more interesting `solution with targets` edge, representing the case in which the solution provided by the solver foresees the involvement of other domain objects, which are firstly found (`Targets Found` state), and then triggered (`Issues Targeted` state) to be involved in the resolution process. Once the current domain object receives feedback from the triggered domain objects (`Solution Received` state), it selects the best solution (`Solution Chosen` state) (e.g., by applying the approach in [158, 159]). At this point, we should distinguish two cases. If the issue was triggered internally (`root node` edge), the domain object first asks the involved targets to commit their local best solution (`Ask Partners To Commit` state); then it waits for their commit to be done (`All Partners Commit Done` state), and eventually it commits

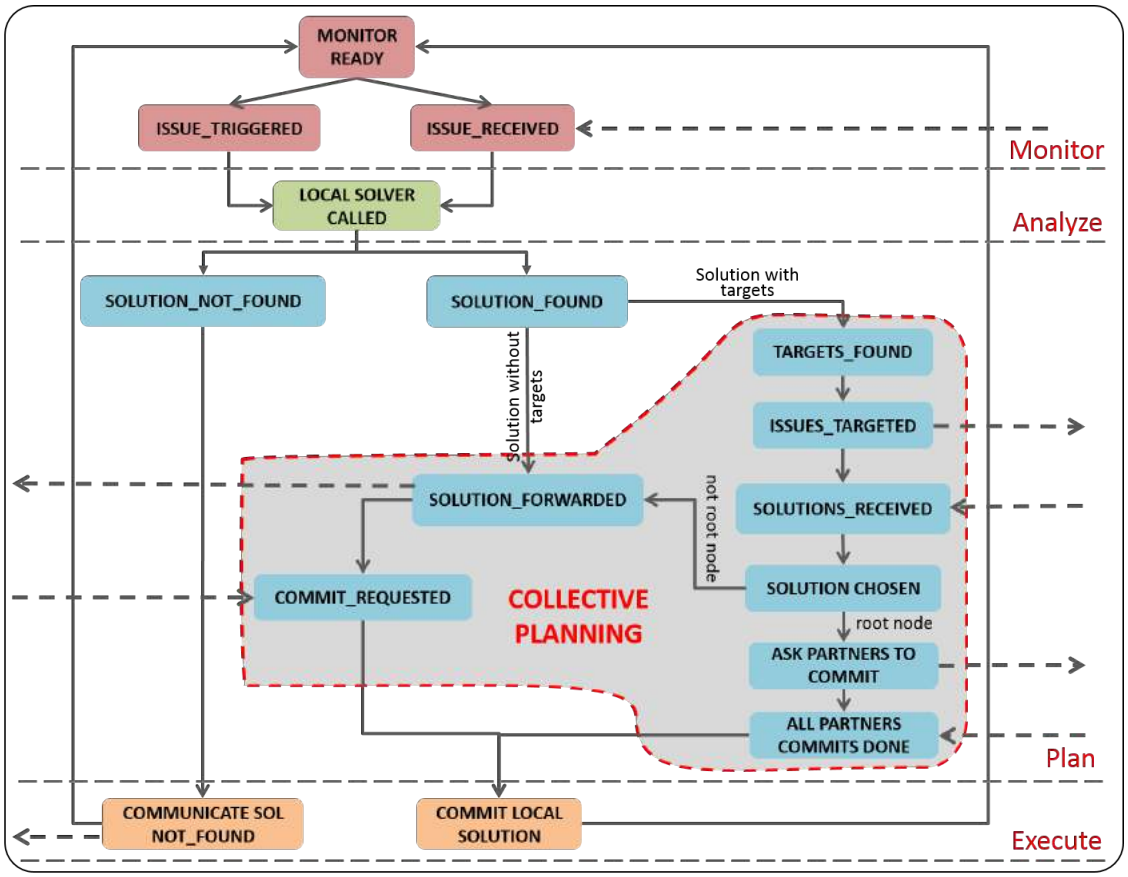


Figure 6.8: MAPE State Machine.

its local solution (`Commit Local Solution` state). Otherwise, if the issue was coming from outside (`not root node` edge), the domain object reports the feedback to the issues sender (`Solution Forwarded` state), and then it waits for a future commit (`Commit Requested` state). The domain object can receive a positive or a negative reply for its proposed solution. In both cases, it executes a solution commit (`Commit Local Solution` state), which will results to be empty in the negative case.

In conclusion, domain objects are dynamically connected also through their MAPE loops interacting to solve collective adaptation problems.

6.3.3 Collective Adaptation Algorithm

In this section we give the pseudo-code definition of our *Collective Adaptation Algorithm* that is supported by the MAPE loops, handlers and solvers

used in the domain objects modeling. We focus on the main procedures, and we give the point of view of a single domain object whose monitor captures an issue. The issue resolution process generates a resolution tree, modeling all possible solutions for a given issue, over which the best solution is selected. First of all, the function `startCA` (lines **1-7**) takes the detected issue and calls the main function `resolveIssue`, by passing to it both the issue and the entity. After the issue resolution process, if a collective solution has not been found, the entity will self adapt (line **5**). The `resolveIssue` function (lines **8-36**) is called locally by the domain object detecting the issue. If needed, the function is recursively invoked to trigger the issue resolution across multiple domain objects in a distributed way. Indeed, the domain object initially calls its solver for the specific issue, if any (line **10**). The called solver provides a solution, which may comprise, in turn, one or more sub-issues (line **11**) that are triggered as a consequence of the first one.

In order to solve each sub-issue, the domain object must establish one or more communications with other connected domain objects (line **12**). For each communication, the set of potential solvers is identified through all reachable domain objects (e.g., targets) (line **16**). Once all targets have been identified, to understand if and how they can handle the sub-issue, the `resolveIssue` function is called remotely on each target by a remote procedure call (rpc) (lines **17-20**). When the domain object receives all the potential solutions for the invoked targets, the Analytic Hierarchy Process (AHP) algorithm [158, 159] is executed to identify the best solution (line **22**). If a best solution has been identified, we can observe two different behaviors: (i) if the domain object running the algorithm is not the root (i.e., the one from which the resolution process started), it stores the solution locally (line **26**) and it waits for a commit request coming from the domain object that is its father in the issue resolution tree. Otherwise, (ii) the root domain object executes the commit of the best solution (line **28**). Since the best solution is implicitly made of sub-solutions related to the different entities involved, the `commit` function (lines **37-42**) acts as follows. Given the tree-path corresponding to the best solution, the root domain object asks to all the targets on this path to commit and execute their sub-solution (lines **38-40**). Lastly, it commits the best solution (line **41**)

and executes it, by ending the issue resolution process.

Algorithm 1 Collective Adaptation Algorithm.

```
1: function startCA(issue)
2:     if resolveIssue(issue, this) then
3:         Issue solved via CA.
4:     else
5:         this.selfAdapt
6:     end if
7: end function

8: function resolveIssue(issue, entity)
9:     solutionFound = False
10:    solution = callSolver(issue)
11:    for all issue ∈ solution.issues do
12:        Coms = deriveComs(issue)
13:        for all comm ∈ Coms do
14:            S = ∅
15:            bestSol = null
16:            T = findTargets(comm)
17:            for all target ∈ T do
18:                S = S∪
19: rpc(resolveIssue(comm.issue, target))
20:            end for
21:        end for
22:        bestSol = AHP(S)
23:        if bestSol! = null then
24:            solutionFound = True
25:            if entity != root then
26:                store(bestSol)
27:            else
28:                commit(bestSol, T)
29:            end if
30:        else
31:            this.selfAdapt
32:            return
33:        end if
34:    end for
35:    return solutionFound
36: end function

37: function commit(bestSol, T)
38:    for all target ∈ T do
39:        execute(target.bestSol)
40:    end for
41:    execute(bestSol)
42: end function
```

In Figure 6.9, we give an example of the communication between two single entities. We show the possible flow of information when an issue (e.g., `RouteABlocked`) is raised internally by an entity (e.g., `Flexibus Driver`). This is a simplification for presentation purposes with the aim of showing the type of communication and synchronization between entities. We have to consider that in normal scenarios our framework is able to deal with multiple entities that collaborate to solve issues.

The labels with ordered numbers represent the order in which the different computational states of the respective state machines depicted in Figure 6.8 are executed. Focusing on the `Flexibus Driver`, in the *Analyze* phase he selects the solver `RouteA-Blocked` able to solve the issue triggered during the *Monitor* phase. In the *Plan* phase, and more precisely in the

`Issue Targeted` state, the solution generated by the selected solver includes the triggering of an extra issue (e.g., `NotifyRouteABlocked`) captured in the *Monitor* phase of the `Route Manager`, in its `Issue Received` state. The `Route Manager`, through its *Analyze* phase finds the right solver, `ManageRouteA` able to solve the captured issue. The solution generated by the solver is forwarded to the `Flexibus Driver` during the *Plan* phase. At this point the `Route Manager` goes in the state `Solution Forwarded`. At this stage, the *Plan* phase of the `Flexibus Driver` is resumed; after an internal reasoning, it will move in the state `Commit Request`. This means that the `Route Manager` is asked

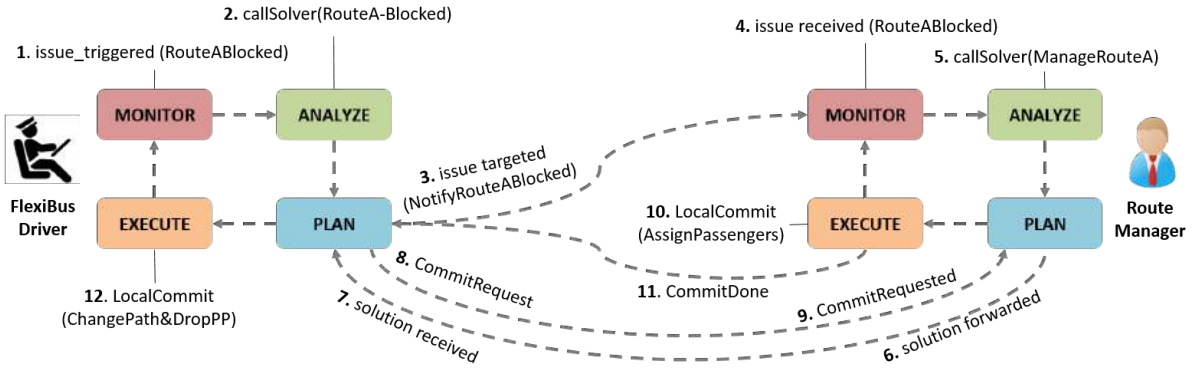


Figure 6.9: Overview of the Communication between two Entities.

to commit the previous calculated solution (`Commit Requested` state). The `Route Manager` proceeds with its *Execute* phase where the `Local Commit` and the `Commit Done` states are executed. Finally, when the `Flexibus Driver` receives the commit notification from the `Route Manager`, he simply needs to start the *Execute* phase where the `Local Commit` state is reached.

6.4 Formal Framework

In this section, we introduce the formal model of our approach for collective adaptation. In section 6.4.1 we formalize the concepts of role and ensemble, while in section 6.4.2 we discuss how to handle issues and communications.

6.4.1 Roles and Ensembles

As we have described in section 6.3, our model of collective adaptation is built around the concept of *ensemble* that represents a collection of autonomous entities collaborating to perform certain tasks.

We introduce the notion of the entity to represent actor(s) capable of playing multiple roles in different ensembles at a given time (e.g., a *Person* entity can be a passenger or a driver in a Flexibus Route ensemble).

Definition 20 (Entity)**.** *An entity is defined by a set of roles it can play,* $\epsilon = \langle R \rangle$.

A role that an entity can play in an ensemble is given by the ways it collaborates with other roles. Collaboration primarily means taking *actions* that change the state of the world. A role also defines a set of *issues* that it can produce, i.e., formation of critical situations. When an issue arises from any sources, a role can choose to handle the issue using one of its individual *solvers*. Finally, a role can define *preferences* that determine actions it is willing to take. A Role Type is defined as follows:

Definition 21 (Role Type)**.** *A role type is a tuple* $r = \langle rA, rS, U, S, rP \rangle$*, where:*

- rA *is a set of action types defined for the role;*
- rS *is a set of variable types that define the current state;*
- U *is a set of issue types that can be produced by a role;*
- S *is a set of solver types that are provided by a role;*
- rP *is a list of preference types available for a role.*

For example, the *Passenger* role type in a Flexibus Route ensemble can be defined as follows:

- $rA = \{\mathsf{embark}(p, c, l), \mathsf{debark}(p, c, l), \mathsf{walk}(p, l_1, l_2)\}$;
- $rS = \{\mathsf{at}(p, l), \mathsf{in}(p, c)\}$;
- $U = \{delay, cancellation\}$;

- $S = \{findWalkingRoute\}$;
- $rP = \{travel\ time,\ travel\ cost,\ walking\ distance\}$.

The action and variable types are introduced later on. Issues are referred to things that can go wrong. The primary way for a passenger to handle any issue in our simplified scenario is to find a walking route to his or her target location. Finally, each passenger's preference towards the time, cost or walking distance may vary.

Definition 22 (Role Instance)**.** *A role instance is a tuple* $ri = \langle ri_{id}, r, A_{ri}, V_{ri}, s_{ri}, P, \Uparrow_r \rangle$*, where:*

- ri_{id} *is the role instance identifier;*
- r *is a role type;*
- A_{ri} *is a set of instantiated actions for the role instance;*
- V_{ri} *is a set of instantiated variables for the role instance;*
- s_{ri} *is a state on the set of instantiated variables;*
- P *represents the preferences of a role instance;*
- $\Uparrow_r$ *is a set of active issue resolutions.*

For example, in case of the role instance corresponding to the Route Passenger p_3 of Figure 6.7 in Section 6.3.2:

- The set of instantiated actions A_{p_3} includes all actions that are formed by assigning objects to the action types embark, debark and walk;
- The set of instantiated variables V_{p_3} includes all variables that are formed by assigning objects to the variable types at and in; and
- The state is given by $s_{p_3} = \{\mathsf{at}(p_3, l_2)\}$, i.e. the (current) origin location of p_3.

We now explore the definition of ensembles to get an insight into the collective behavior of multiple roles. Each ensemble describes a certain type of collective behavior that may take place in the domain of interest.

Ensemble includes a set of roles and corresponding constraints. The constraints can generally be of any kind, but we treat them as role cardinality constraints (i.e., intervals that bound the number of role instances for each role):

Definition 23 (Ensemble Type)**.** *An ensemble type e is a set of role types R.*

For example, a Flexibus Route ensemble type can be defined as $Flexibus\ Route = \{Passenger, FBDriver, FBCompany\}$, where $FBDriver$ is the role type defining Flexibus drivers, and $FBCompany$ is the role type defining Flexibus companies. An ensemble instance is formed by assigning multiple role instances to each role type, e.g., a Flexibus Route ensemble instance can have multiple passengers and Flexibus drivers.

Definition 24 (Ensemble Instance)**.** *An ensemble instance is a tuple* $ei = \langle ei_{id}, e, RI \rangle$*, where:*

- ei_{id} *is the ensemble instance identifier;*
- $e = R$ *is an ensemble type;*
- RI *is a set of role instances of types in* R*.*

In the next section we explore the definition of issues and communication.

6.4.2 Issues and Communication

In addition to finding joint plans, collaboration also consists in producing issues and taking care of issues produced by others. As such, a role includes a set of issue types it can produce, generally correspond to different critical situations that can happen to a role of an ensemble. Each issue type includes a set of parameters describing it:

Definition 25 (Issue Type)**.** *An issue type is defined by a set of parameters* $u = uP$*.*

For example, a Flexibus driver can trigger an issue type $routeDelay = delayTime, delayReason$. An issue instance corresponds to a particular critical situation occurring in an ensemble (e.g., a flexibus delay that happened to a particular driver at a particular moment in time with a certain number of passengers waiting or already on-board). It belongs to an existing issue type and its state is determined by values assigned to issue parameters:

Definition 26 (Issue Instance)**.** *An issue instance is a tuple* $ui = \langle u, L_u \rangle$*, where:*

- u *is an issue type;*
- $L_u : u.uP \rightarrow V$ *is an assignment function for issue parameters.*

Each role, collaborating in an ensemble can provide one or more solvers. In our formal framework, we have two types of activities that a role instance can execute during a collective adaptation problem resolution: *issue communication* and *issue resolution.* Issue communication is used to send an issue instance to be solved to a *target role instance* (see definition below). The issue instance may be sent to multiple partners at a time in an attempt to find a better solution. Issue communication comprises a few steps: 1) the issue is sent to all target roles; 2) the replies are received from the partners, who could resolve the issue; 3) the preferable solution is chosen; and 4) the preferable solution is committed. Formally, a target role instance and issue communication are defined as follows:

Definition 27 (Target Role Instance)**.** *A target role is a tuple* $t = \langle ri_{id}, s_i, p \rangle$*, where:*

- ri_{id} *is the target identifier (role instance id);*
- s_i *is the solver instance invoked to solve the issue* $s_i.ui$*;*
- p *is the solution proposed by the target. It is a process that the target role will execute if it will become part of the overall issue resolution.*

Definition 28 (Issue Communication)**.** *An issue communication is a tuple* $\uparrow_u = \langle ui, T \rangle$*, where:*

- *ui is an issue instance communicated;*

- *T is a set of target roles.*

While the issue communication is a way to propagate resolution activities between partners, *issue resolution* corresponds to the high-level model of internal elaboration being done by role instances. In particular, we assume that the issue instance may either arise internally (when the issue originally occurs in this role instance) or is received by one of the role instance solvers. As soon as the issue instance is raised, the role instance may either resolve it locally or send one or a few issues to the other partners as a part of the resolution procedure. The issue resolution is formally described as follows:

Definition 29 (Issue Resolution)**.** *An issue resolution is a tuple $\uparrow_r = \langle ri_{id}, ui, \Psi \rangle$, where:*

- *ri_{id} is the identifier of the role instance, from which the issue instance arrived (null if the issue arouse internally);*

- *ui is an issue to be resolved;*

- *Ψ is a set of alternative solutions, each is a tuple $\psi = \langle ui, \Uparrow_u^O, p_{ext}, p_{int} \rangle$ where ui is the issue to be resolved, $\Uparrow_u^O$ is a set of outgoing issue communications, p_{ext} is a process (solution) that is supposed to be sent to the role instance associated with ri_{id}, while p_{int} is the internal process (solution) for the specific issue instance arrived.*

If the resolution is fully local, in each solution $\psi \in \Psi$ the set of $\Uparrow_u^O$ of the outgoing issue communications is empty. Otherwise, $\Uparrow_u^O$ correspond to the communication of all subissues that must be resolved in order to resolve the original issue.

The focus here is on collaborating through actions, either when computing an initial shared plan, or when re-planning as a result of issues triggering.

6.5 Evaluation of the Collective Adaptation Approach

The collective adaptation approach, based on the extended design for adaptation framework, has been designed, implemented and executed in the context of the ALLOW Ensembles [35] project's demonstrator , called ***DeMOCAS*** [31]. We will present DeMOCAS in chapter 7, which relates to the implementation and evaluation of the design for adaptation approach presented in this dissertation. In this section, instead, we specifically report the evaluation of the collective adaptation algorithm just presented in this chapter. We realized a Java implementation of the algorithm, and we have evaluated it by executing experiments in the two different scenarios described in section 6.2. In our experiments we are concerned with measuring and understanding how the framework performs in providing solutions in terms of collective adaptation. Thus, we do not focus on the correctness and complexity of the solutions provided by each solver of an entity, which are usually domain specific, but on the evaluation of our collective adaptation approach. In the following we present the experiment design and the discussion of its results.

Experiment Design.
The main goal of our evaluation is to analyze the framework with respect to its *feasibility* and *scalability* in the context of the application scenarios introduced in section 6.2. This goal can be refined into the following research questions:

- **RQ1:** Can the framework be used at run-time to manage the adaptation of service-based Collective Adaptive Systems?
- **RQ2:** Is the framework scalable for managing real-sized applications?

In our experiments we perform a stratified random sampling [160]: the population of all possible sequences of raised issues is divided into a set of *treatments* with a uniform distribution between the groups in terms of the number of raised issues. Random sampling is then applied within the groups. Each *treatment* models a sequence of raised issues. More specifically, in each treatment t we can have a set of different raised issues with one of the following cardinality: $< 1, 250, 500, 750, 1000 >$. Moreover, both

the order of the raised issues within the sequence and the entities raising them is randomly chosen. As an example, considering the UMS scenario, the value $< 250 >$ represents the treatment in which the total number of different raised issues (e.g., `RouteBlocked`, `PassengerDelay`, etc.) sums up to 250 issues in total. After the treatments generation, the experiment has been run.

Discussion of the Results.
The specification of the UMS scenario we used to evaluate our approach contains 8 ensembles models and 23 domain object models, while the SurSys scenario contains 5 ensembles models and 24 domain object models. We have evaluated our techniques using a dual-core CPU running at 2.7GHz, with 8Gb memory. We created 500 treatments for each scenario resulting in a total of 1000 runs of the experiment.

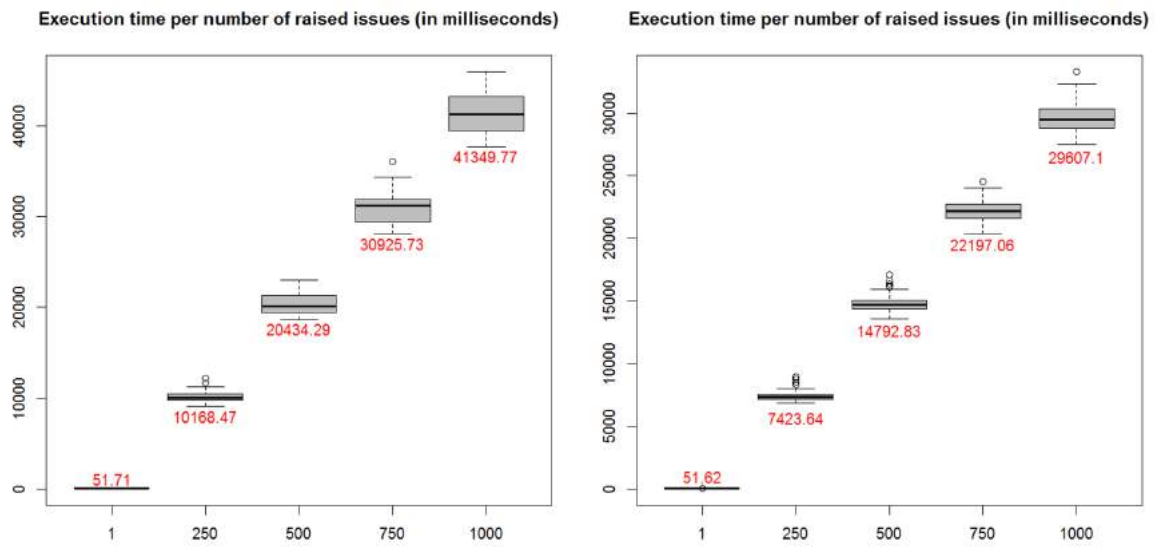


Figure 6.10: Execution Time per number of raised issues (UMS in the left side).

Figure 6.10 shows the average execution time per number of raised issues, after a sequential execution of 500 treatments, for both the scenarios. We can observe that after a full loaded execution, our framework can solve a burst of 1000 issues in under 42 and 30 seconds (s), in the UMS and SurSys scenarios, respectively. However, it is important to consider that having burst of 1000 issues all together is very rare.

In order to better understand how the framework performs when the complexity of the collective adaptation problems increase, we did as follows. We consider the number of roles involved in the issue resolution as the complexity index for each problem. Then, we take into account the subset of treatments of 500 issues, to evaluate the performance on a set of equivalent problems. In Figure 6.11 and Figure 6.12 we report the obtained results. For the UMS scenario (see Figure 6.11), we can see that the considered set of problems involves a number of roles in a range between 4 and 9, while for the SurSys scenario (see Figure 6.12) the range is between 7 and 11. This shows that the problems complexity is almost uniformly distributed. We can notice that the average execution time moderately increases with respect to the increasing of the problem complexity, but not in a linear way. In the UMS scenario we can observe a particular increase of the execution time for the problems with complexity 9. However, by analyzing the data, we observed that in a set of 2500 issues resolutions, only one problem involves 9 roles reducing the performance.

Concerning the *scalability of the framework* in real-size problems, we decided to measure the distribution of the total number of issue resolutions both over the ensembles and roles number, in the different scenarios. As shown in Figure 6.13, the number of ensembles is between 1 and 4. In the UMS scenario, for the majority of the problems, the issue resolutions happens in the scope of 2 ensembles. While, for the SurSys scenario, the majority of the problems is solved by involving 3 ensembles. Observing the distribution of the issue resolutions over the roles, as reported in Figure 6.14, we can notice that the trend is almost equivalent to the distribution measured over the ensembles. In the UMS scenario, in most cases the issue resolution involves 5 or 6 roles. As regard the SurSys scenario, this number increases around 9 and 10. These results clearly demonstrate the scalability of our framework when dealing with real-sized applications.

In conclusion, to demonstrate the *generality of our approach*, we considered two large-scale and distributed systems that typically use two different decision management strategies. The UMS is hierarchical by nature, while the SurSys follows both a peer-to-peer and a hierarchical communication strategy. The experiments showed that our framework performed well while dealing with both the scenarios. Moreover, by looking at the

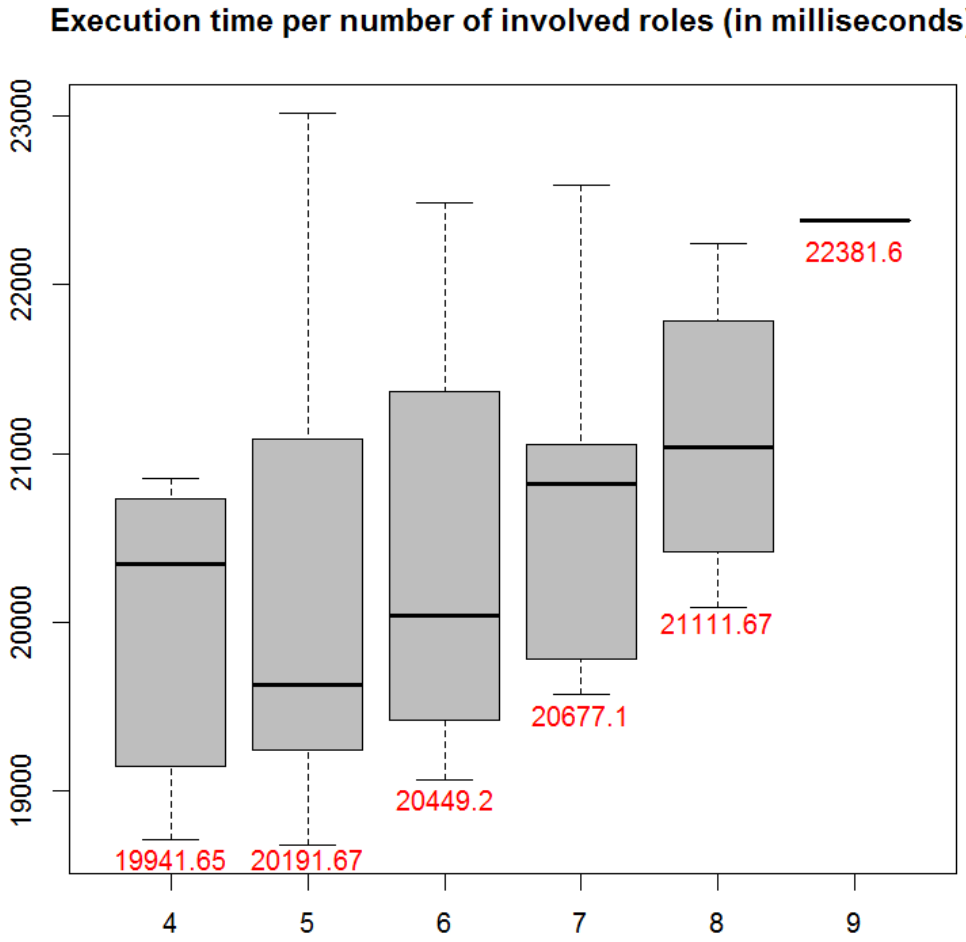


Figure 6.11: Execution Time per number of involved roles for the UMS.

resulting graphics it is possible to clearly distinguish between the different decision management strategies used by the two different scenarios. For instance, as emerges from Figure 6.13, in the UMS scenario the system is often able to solve the problem within the ensemble in which it arises (e.g., *intra-ensembles* resolution). To the contrary, in the SurSys scenario, the system always solve the problem in the scope of more than one ensemble (e.g., *inter-ensembles* resolution). Furthermore, Figure 6.14 reflects the strategy used by the systems, by showing that systems with a hierarchical strategy, as the UMS, involves a less number of roles, with respect to

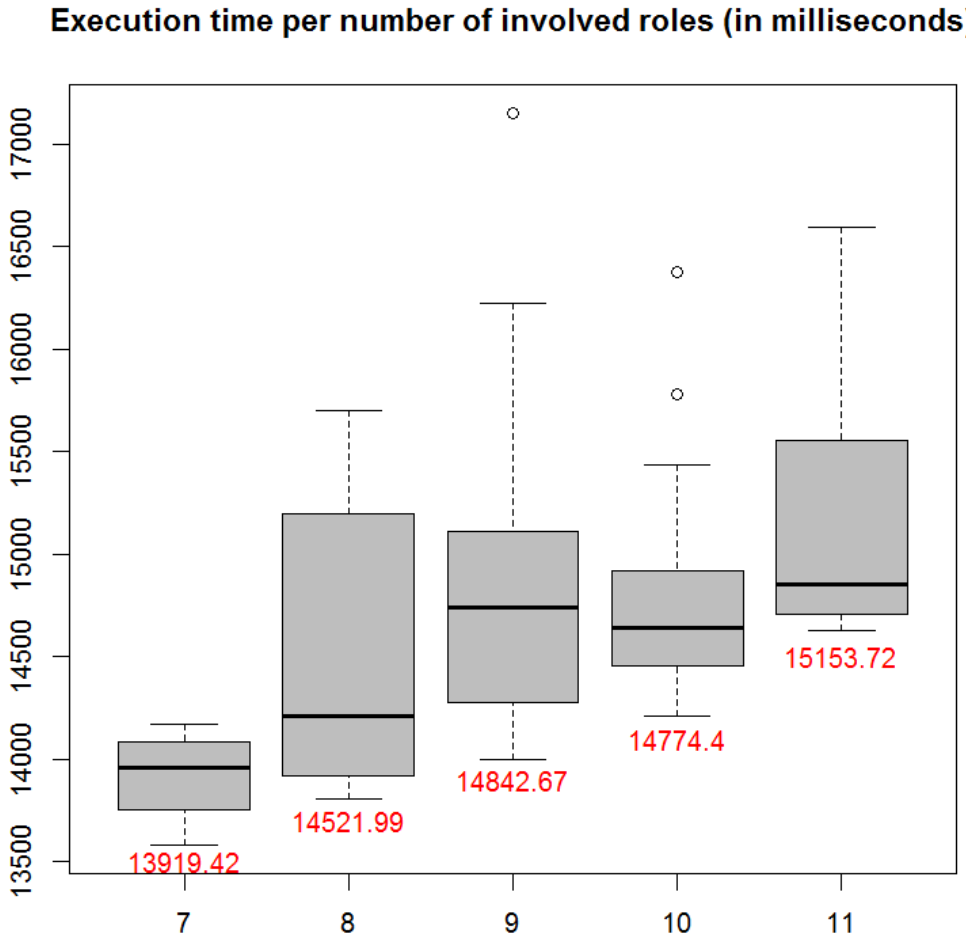


Figure 6.12: Execution Time per number of involved roles for the SurSys.

those with a peer-to-peer strategy, as the SurSys. For interested reader, we provide a *replication package* of our experiments[2].

6.6 Discussion

In this chapter we addressed the challenge of collective adaptation in service-based CAS, by exploiting the design for adaptation approach pre-

[2] **Replication Package.** To allow the easy replication and verification of our experiments, we provide a complete and portable replication package, which is publicly available at the link `https://github.com/das-fbk/CAS-ICSOC2016`.

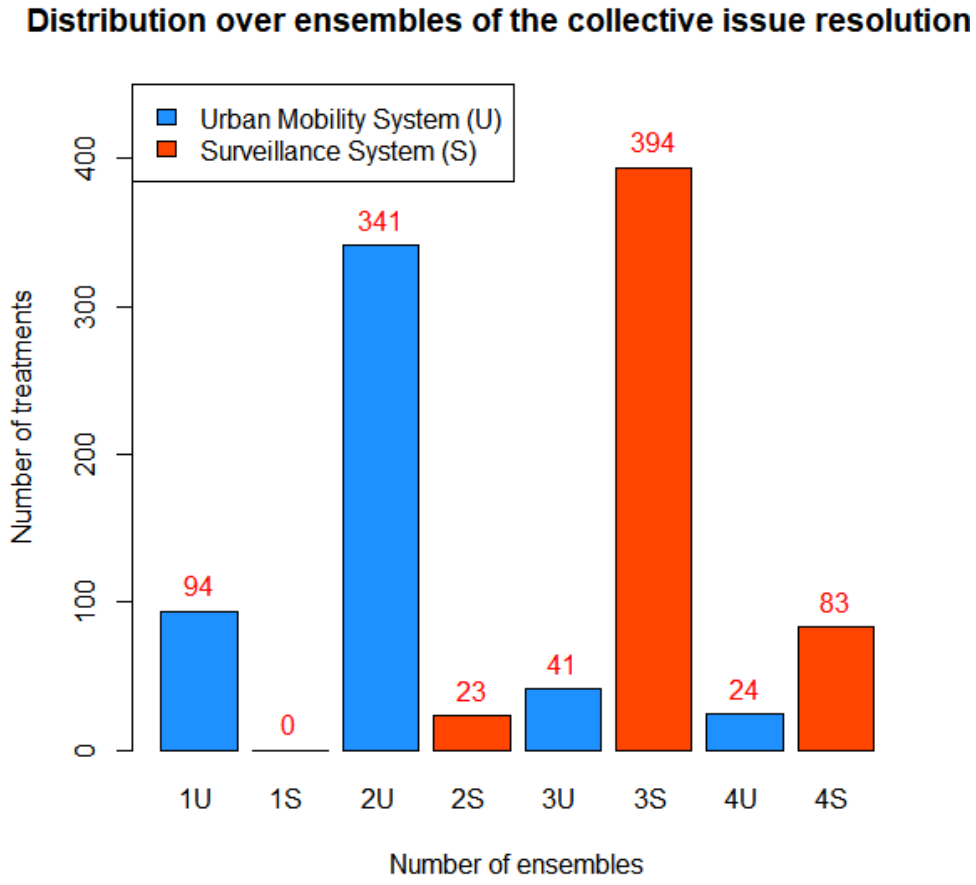


Figure 6.13: Distribution of Issue Resolutions over the number of Ensembles.

sented in this dissertation to support the modeling, development, and execution of CAS operating in dynamic environments. Key properties of our approach are **R1**) the capability of handling with an open and dynamic system environment, such as the mobility domain environment; **R2**) the support for different decision management strategies, from peer-to-peer (e.g., SurSys) to hierarchical (e.g., UMS) models; **R3**) the emphasis on a collaborative, yet distributed, management of adaptation problems among entities, as shown by the Collective Adaptation algorithm, also supported by the MAPE loops equipping each domain object (i.e., entity) in the system.

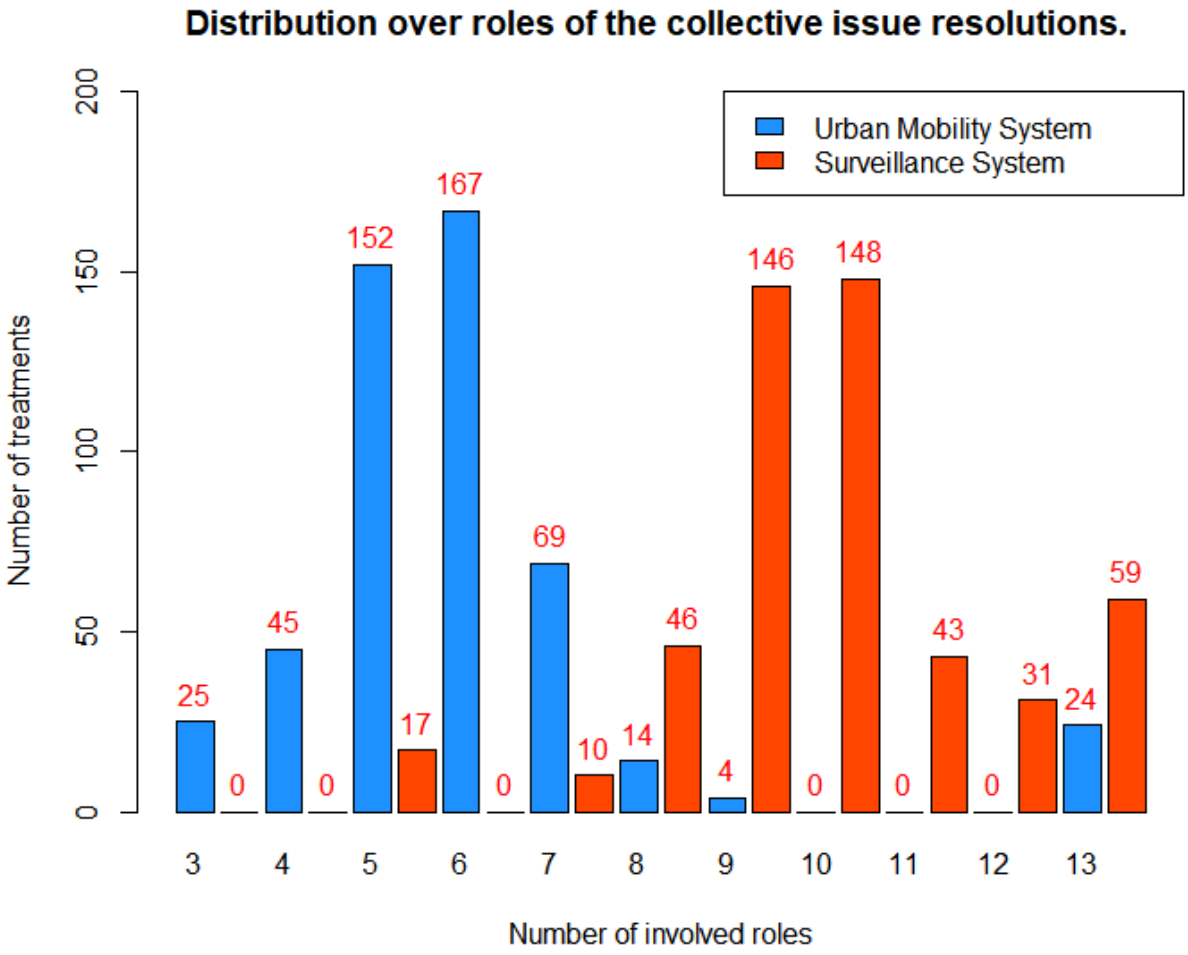


Figure 6.14: Distribution of Issue Resolutions over the number of Roles.

Following this principle, we can say that our framework supports the design, development, and operation of CAS that are *resilient* to a wide range of changes.

The defined *Collective Adaptation algorithm* is able to solve adaptation issues within and among ensembles, which discover the entities able to solve an issue and to apply adaptation with minimal impact, by guaranteeing the system reliability.

Lastly, both *intra* and *inter-ensembles* adaptation can be performed with our approach.

In contrast to the related work analyzed in section 6.1, our approach gives the possibility to handle both *cooperative* and *selfish* behavior between entities. Furthermore, we apply a *decentralized management of coalition*, thus eliminating the single point of failure and potential bottlenecks in the system. As last, as shown by the examples, our approach provides support for *run-time adaptation* both in mission critical and safety critical systems. In summary, comparing to the other works, which focus only on one or few criterion, we can say that our approach is multi-criterion with respect to the aspects highlighted in the final discussion in section 6.1. The potentialities of our solution can be summarized in four main characteristics:

- **Generality:** as already said, our approach is general, in the sense that it is not domain or problem specific. This gives us the opportunity to reuse it in different domains, as we did in this chapter, by applying it at two different scenarios: the Urban Mobility Systems and The Surveillance Systems.

- **Heterogeneity:** key properties of our approach are the emphasis on *collaboration towards fulfillment of individual diverse goals* and the heterogeneous nature of an ensemble with respect to roles, behaviors and goals of its participants. Indeed, differently from *swarms* [154], our ensemble model do not require to the entities to have a uniform behavior and a global shared goal.

- **Scalability:** alternative adaptations may be available, and given the multiple criteria and preferences of the different entities involved in an ensemble, such adaptations may need to be ranked. To this aim, in our framework we exploit a multi-criteria ranking approach, based on analytic hierarchy process (AHP) [158] [161] that allows the selection of the best adaptation alternatives with respect to the preferences of the entities involved. Moreover, our approach supports a *hierarchical structure* of systems. This is very important because it provides scalability and allows different entities with different knowledge to take decision at different levels.

- **Openness and Incrementality:** our solution enables systems with

collective adaptability to be built as *emergent aggregations* of autonomous and self-adaptive entities. Ensembles can be created spontaneously and change over time: indeed, different entities may decide to join and/or leave an existing ensemble dynamically and autonomously. Termination of ensembles may also be spontaneous. It may occur when some or all entities, involved in an ensemble, leave because, for instance, they have reached their goals, through collaboration.

As we said before, the collective adaptation approach described in this chapter has been implemented and evaluated in the DeMOCAS [31] demonstrator. In particular, DeMOCAS provides also the complete implementation of the UMS as a CAS, by fully exploiting the extended design for adaptation approach introduced in chapter 6. Thus, we refer to the next chapter 7 for more details about the UMS scenario, its usage, its design and its execution and collective adaptation.

For what concern future work about our collective adaptation in service-based systems approach, we plan to use our framework in real environments. In particular, regarding the *Surveillance System*, our idea is to integrate it with a suitable extension of the FLYAQ platform [162]. This platform permits to graphically define civilian missions for a team of autonomous multi-copters via a domain specific language. With respect to the *Urban Mobility System*, instead, a future step would be that of integrating it with an extension of the `Open Trip Planner` [3] to provide not only the multi-modal trip planning solutions but also collective mobility solutions (e.g. car pooling, flexi-buses).

[3] `http://www.opentripplanner.org/`

Chapter 7

Implementation and Evaluation

With a knowledge of the models, execution handlers and adaptation mechanisms that comprise our approach (see chapters 4, 5 and 6), together with the life-cycle process we can use to shape adaptive service-based systems (see chapter 3), all that we need is an implementation framework allowing us to pull these pieces together. On top of this framework we realized two demonstrators, namely ATLAS [29] and DeMOCAS [31]. In particular, ATLAS is the implementation of the Travel Assistant representing the motivating scenario of this dissertation, while DeMOCAS is a simulator for collective adaptive systems, currently implementing and executing the Urban Mobility System mentioned in chapter 6. Moreover, to see these demonstrators in action, or simply inspect the full specification of the travel assistant and the urban mobility system, one can freely download both ATLAS [1] and DeMOCAS [2].

This chapter is specifically devoted to the implementation and evaluation of our design for adaptation framework, performed through the realization and execution of ATLAS and DeMOCAS. Moreover, although we provide approaches for the design and operation of (collective) adaptive systems, as well as techniques for performing both *individual* and *collective* adaptations, a sound engineering process for CAS is still missing. In particular, a CAS is specified at a low level of abstraction (XML files), a task that tends to be time-consuming and error-prone when the size of the system grows. Therefore, we extended our framework by specifying a domain specific language (DSL) for defining CAS, named CAStlE [32].

[1] ATLAS is available at: `https://github.com/das-fbk/ATLAS-Personalized-Travel-Assistant`.
[2] DeMOCAS is available at: `https://github.com/das-fbk/DeMOCAS`.

Consequently, this chapter is mainly founded on the work presented in [29], [31] and [32] and it further extends them. The rest of the chapter is structured as follow. In section 7.1 we give an overview on the design for adaptation framework and its implementation. Section 7.2 is devoted to the description and evaluation of ATLAS, while section 7.3 reports details about DeMOCAS and its evaluation results. CAStlE, instead, is reported in section 7.4. Finally, we close the chapter with a final discussion in section 7.5.

7.1 Design for Adaptation Framework

The design for adaptation framework has been implemented by using ***Java*** as programming language. Then, we used ***Eclipse*** as developing IDE and ***Maven*** for the dependencies development and the project organization. The component diagram of the framework is reported in Figure 7.1. In particular, for the implementation of the collective adaptation approach described in chapter 6, specific components have been developed to design, implement and deal with collective adaptive systems and the execution of their collective adaptation. These components are green-colored in Figure 7.1. The framework is composed by five modules, namely the ***Execution***, ***Adaptation***, Artificial Intelligence ***(AI) Planning***, ***Presentation*** and ***Model*** modules.

The ***Presentation*** module implements different components making the Graphical User Interface (GUI) of the framework, together with the functionalities allowing the visualization of the execution of running scenarios of service-based systems. These components can, then, be used in the interfaces of the demonstrators developed on top of the framework. We will see further on in this chapter the different user interfaces of ATLAS and DeMOCAS, which we are going to describe. In particular, in ATLAS, which effectively implements the travel assistant, the user interacts with a Telegram chat-bot [3] acting as the main interface, while a connected demonstrator shows the execution of the travel assistant process and its dynamic specializations, which cannot be observed on the Telegram chat-bot. In DeMOCAS, instead, which is a simulator for CAS and which implements

[3] `https://telegram.org/`

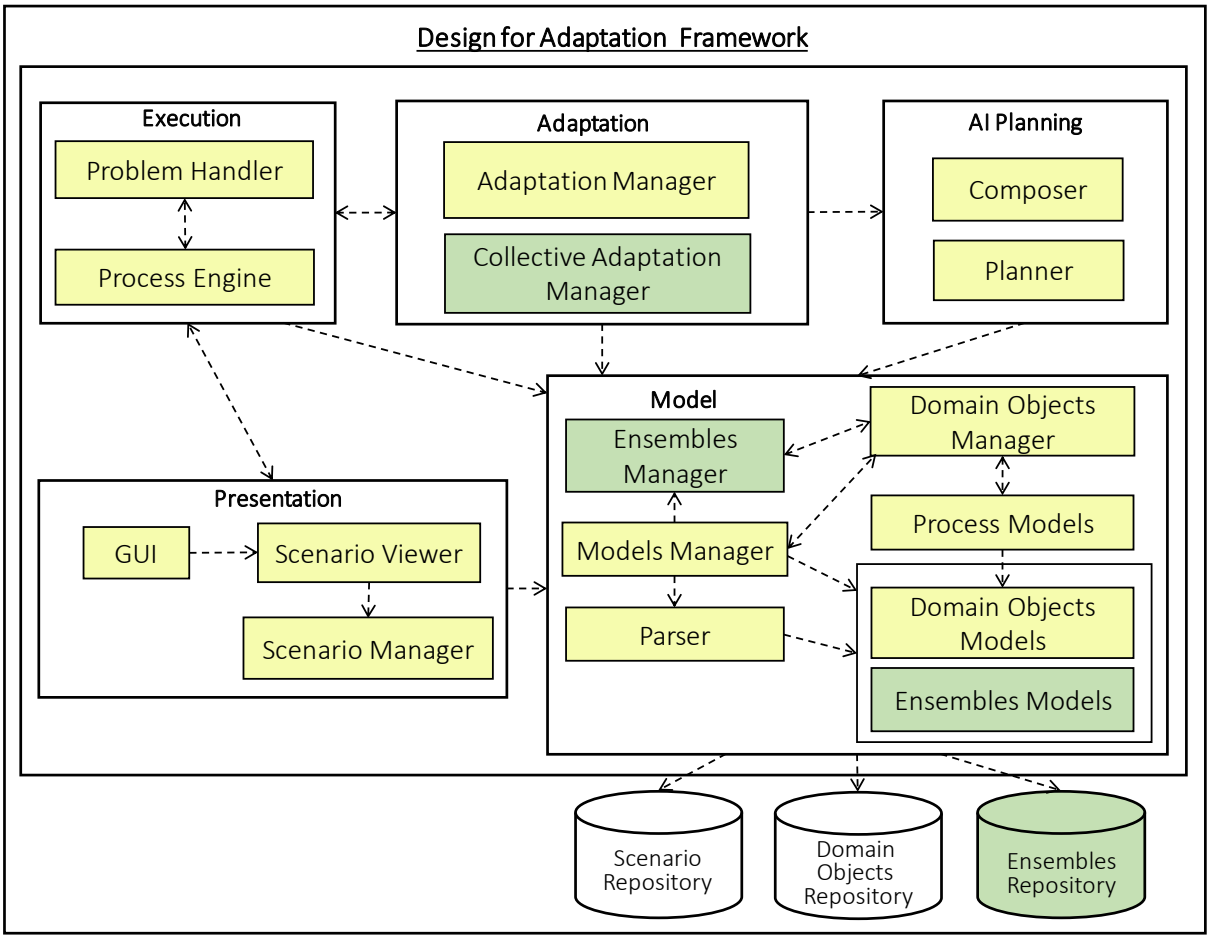


Figure 7.1: Design for Adaptation Framework Component Diagram.

the urban mobility system scenario, the demonstrator's interface is extended with a map, on which the user can easily follow the evolution of the running scenario. The Presentation module is constantly synchronized with the Execution module, from which it obtains the details about the scenario execution.

In the ***Model*** module, the *Parser* is responsible for loading the scenario files (i.e., the Domain Objects and its constructs, the Ensembles and other scenario related files) from the file system and parse them. The *Process Models* defines the basic building blocks for a process definition (e.g., input, output, concrete and abstract activities), while the *Models*, *Domain Objects* and *Ensembles Managers*, are essentially devoted to handle the scenario models, domain objects and ensembles life-cycles.

The ***Execution*** module contains the components for executing adapt-

able processes. The *Process Engine* is in charge of performing the execution of the core processes of the running domain objects instances, by executing all the corresponding activities. When run-time problems occur, the *Problem Handler* creates requests for the Adaptation module.

The ***Adaptation*** module implements all tasks related to process adaptation. It is called by the Process Engine. When it must tackle the refinement of an abstract activity (*local* adaptation), as well as an issue resolution (*collective* adaptation), it uses the *Adaptation Manager* and the *Collective Adaptation Manager*, respectively. The Adaptation Manager is responsible for deriving the planning domain by driving the fragments selection and ranking (see [128]), according to the goal of the abstract activity and the specific execution context. The Collective Adaptation Manager runs the collective adaptation algorithm, described in chapter 6, for the issue that has been caught, by triggering the issue resolution procedure spanning different domain objects and ensembles.

The ***AI Planning*** module sustains the local adaptation of domain objects, by supporting the refinement of abstract activities. It is in charge of managing the refinement procedure as an AI planning problem (*Planner*), providing the (composition of) fragments to be injected (*Composer*).

Process Engine Implementation. In the implementation of our framework, the *Process Engine* enables the execution of automated and adaptable processes, modeling the behavior and functionalities of domain objects. Before ending this section, we want to briefly discuss about Workflow Management Systems (i.e., process engines), in relation to the context of the work presented in this dissertation.

Before the implementation started, we looked for a process engine suitable for the integration into our design for adaptation framework. To this aim, we performed an investigation on available standard process engines meeting certain conditions, necessary for our purposes. In particular, without going to much into details, we conducted a two-step analysis on a set of process engines. The first step was focused on a selection driven by requirements such as, the license type, the release road-map, the community support, the supported languages and the possibility of extending the process engine with constructs and/or features required in our frame-

work (e.g., abstract activities, process injection). We come out from this first analysis with a subset of eligible process engines, namely *jBPM* [4], *Camunda* [5] and *Activiti* [6]. The second step analysis focused, instead, on more functional requirements. We performed a set of basic tests to observe and understand features, such as, how these process engines deal with the *synchronization* and *communication* among running processes, the *correlation* between processes (i.e., the management of the state of the conversations currently active), the *parallel process execution*, the processes *data* and *variables* management, and the *external events* management (i.e., how a process can send and receive external events). For a detailed and deeper evaluation of Workflow Management Systems (WfMS), we suggest to the interested reader the work in [163]. Among other things, in this work, the authors state, after their evaluation of WfMS, that it does not exist a WfMS addressing all the features that one might be looking for. To the contrary, the selection of a WfMS requires a prioritization approach that ranks the needed features according to their importance for the intended specific usage scenario. Based on the results that we obtained from our analysis of process engines, we completely agree with the authors of [163].

Eventually, after our investigation, we come out with a set of challenges affecting our final decision. In particular, none of the eligible process engines are thought for dealing with (i) the *decentralized management of processes* and (ii) the *correlation among different processes* that are, instead, fundamental in our framework. As a consequence, we decided to realize from scratch a process engine implementing the features required for the execution of service-based adaptive systems realized within our framework. Essentially, it is a conventional process engine, extended with some adaptation-related constructs. The extensions relate to:

- the execution of abstract activities and the injection of (composition of) fragments in place of them;
- the consistency checking to detect conflicting situations requiring the adaptation of a process (e.g., the violation of a precondition);
- the suspension of the execution of a process to apply adaptation, and

[4] https://www.jbpm.org/ [5] https://camunda.org/ [6] https://www.activiti.org/

the subsequent resumption;

- the multiple-instance processes management (e.g., to jump from a process instance to another);
- the correlation and synchronization between processes (e.g., to manage the state of the active conversations);
- a preliminary implementation of the data-flow management allowing processes to exchange data during their inter-operation;

Of course, also our process engine does not address all the typical requirements for executing automated processes. We started by implementing the primary requirements needed for the application of our design for adaptation approach. We intend to further extend the process engine accordingly to the future extensions of the approach (e.g., to add the local adaptation mechanism management).

We conclude this section saying that both the demonstrators that we are going to describe in this chapter rely on the before-mentioned process engine. At this point, given the framework supporting and implementing the design for adaptation approach presented in this dissertation, both the basic and the extended versions, in the next sections we are going to describe, to a greater level of detail, the ATLAS demonstrator in section 7.2, and the DeMOCAS demonstrator in section 7.3.

7.2 ATLAS: a world-wide personAlized TraveL AssiStant

In this section, we introduce ATLAS – *a world-wide personAlized TraveL AssiStant*, a demonstrator developed within our framework and showing how the presented design for adaptation approach supports the development, deployment and execution of adaptive service-based systems operating in dynamic environments. In particular, ATLAS effectively implements the travel assistant of the motivating scenario of this dissertation. Moreover, it relies on the refinement adaptation mechanism (see chapter 5) to

effectively deal with the *dynamicity* and *openness* of the mobility environment, thus with the context-aware specialization of the travel assistant.

We emphasize, here, that for the development of ATLAS we wrapped up *real-world mobility services* exposed as open APIs. Indeed, we exploit ATLAS to further discuss and verify (i) how long it takes to wrap up real services as domain objects, and (ii) how much the automatic refinement (service selection and composition) affects the execution of the travel assistant.

The rest of the section is organized as follow: in section 7.2.1 we run a discussion about the specific challenges of the mobility domain. In section 7.2.2 we discuss how ATLAS is organized as a service delivery platform. Section 7.2.3 is devoted to describe the implementation of ATLAS, while in section 7.2.4 we provide evaluation results as obtained from the design and execution of ATLAS.

7.2.1 Mobility Domain Challenges

In this section, we discuss the challenges related to the mobility domain, which also motivate our work, by providing an analysis on the ecosystem of mobility services.

As already introduced in section 3.1, where we described the Travel Assistant scenario, nowadays, in the mobility context, users dispose of a large offer of mobility services. These services may differ depending on different aspects, such as the offered functionalities, the targeted users, or the provider. In particular, there are *journey planners* (e.g., Rome2Rio, Google Transit) for finding traveling solutions between two or more given locations. Then, *specific mobility services* are those referring to specific transport modes (e.g., CityBikes [7] focuses exclusively on bike sharing data) or provided by transport companies: some examples are Flixbus [8], DB BAHN [9], Trenitalia [10]. Moreover, as already discussed, an emerging trend is that of *shared mobility services* that are based on the shared use of vehicles, bicycles, or other means (e.g., BlaBlaCar). All these services can involve or refer both to *public* and *private* transportation services. Furthermore, mobility services can offer disparate functionalities (e.g., planning,

[7] https://www.citybik.es/ [8] https://www.flixbus.com/
[9] https://www.bahn.de/p/view/index.shtml [10] http://www.trenitalia.com/

booking, ticket payment, seat reservation, check-in and check-out, feedback evaluation, user profiling, and so on). Some functionalities may be peculiar for specific services and/or require particular devices (i.e., the need for unlocking a bike from a rack is peculiar for bike-sharing services, and a smart-card might be needed to do it). In addition, these services are made available through a large variety of technologies (e.g., web pages, mobile applications), with different constraints on their availability (e.g., free vs. pay).

In such a situation, when a user looks for a door-to-door journey plan, to get travel information as more accurate as possible, she must interact separately with a set of different services, to exploit the different functionalities required to accomplish a journey. For instance, if a user wants to go for a vacation in a country other than the one in which she lives, lets say from Verona to Paris, she can use, among many, the following set of services. An inter-modal journey planner (e.g., Rome2Rio) provides a set of alternatives to reach the country. Even in the best case, alternatives are made at least by two legs, that is two transport means (e.g. plane and shuttle). After selecting one option, the user will probably need to search for the specific services offered by the agencies providing the journey (e.g., the plane company and the shuttle company). Now, she must understand how to manage her trip from home to the airport in Verona, and from the shuttle terminal to the hotel in Paris. These are more local/city related transport data. In particular, in Verona the airport is reachable by a dedicated shuttle managed by the ATV agency [11], while in Paris a plenty of local transport services exist (e.g., RATP [12]).

Figure 7.2 shows a snapshot of a representative selection in the ecosystem of mobility services. We define two dimensions, as follow:

- On the *y-axis* we give the *geographic coverage*, ranging from *local* to *global*. It measures the area covered by the mobility services (e.g., city, country, world-wide coverage). For instance, Google Transit is a global planner, since it can be used for planning all around the world. ViaggiaTrento [13], instead, is a local planner, since it combines all the current public transports specifically for the city of Trento, by

[11] `http://www.atv.verona.it/`
[12] `http://www.ratp.fr/`
[13] `http://www.smartcommunitylab.it/apps/viaggia-trento/`

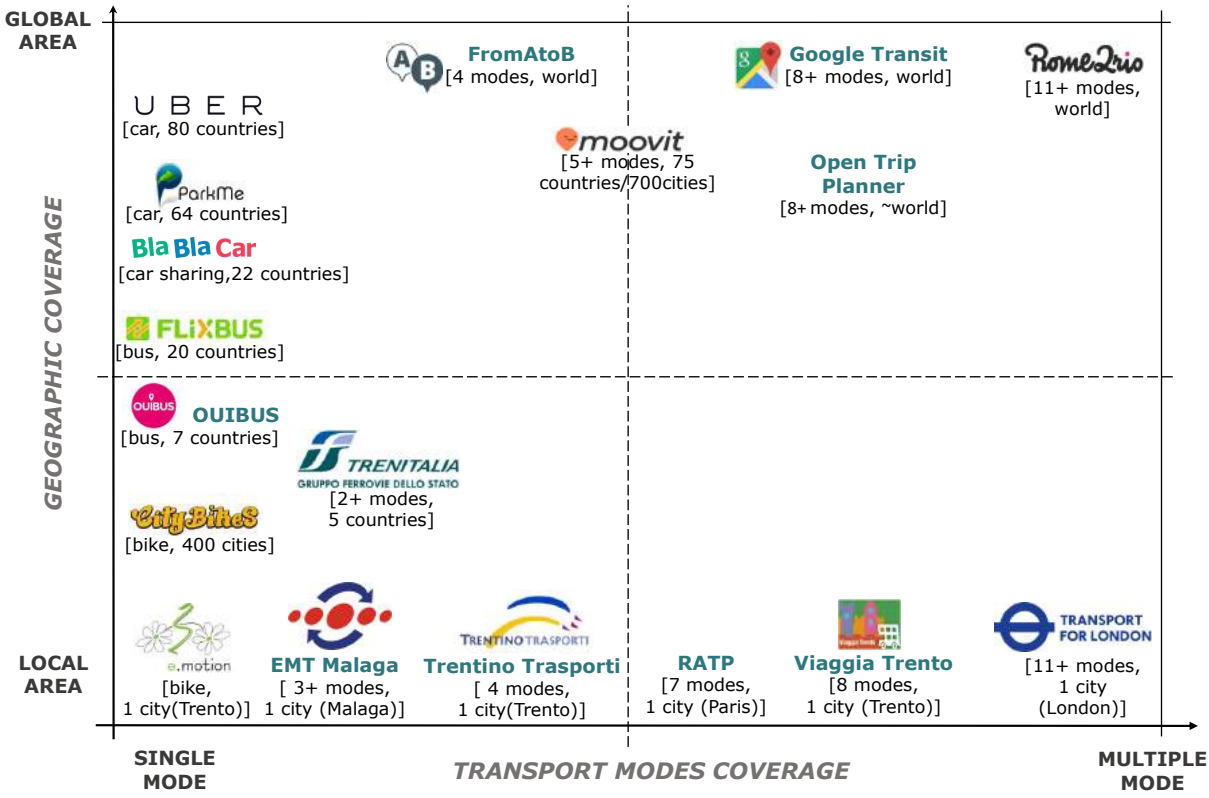


Figure 7.2: Overview of Mobility Services.

providing detailed and accurate plan alternatives.

- On the *x-axis* we give the *transport modes coverage*, ranging from *single mode* to *multiple mode*. It measures the number of different transport means handled by mobility services. For instance, services like Flixbus, e.motion [14] and Uber [15] refer to a single transport mode, namely bus, bike and car. To the contrary, journey planners tipically consider different transport modes, and the same is for (local) public transport company.

Following this classification, we can make several observations about mobility services. In particular, services dealing with one or a few transportation modes, as well as services having a local coverage for one or a few cities/countries (i.e., the services closest to the x and y axis in Figure 7.2) are characterized by a high accuracy of the provided data. To the contrary,

[14] `http://www.provincia.tn.it/bikesharing` [15] `https://www.uber.com/`

the more global are the services, the more they tend renounce accuracy. This can be due to different aspects: for instance, while private companies own and manage their data, other open mobility services can rely on available open data and open API, which can be incomplete or not up to date. If we focus on cities, we can observe that there is a multitude of disparate local services, which are very specific for one or few transport services and very accurate, at the same time. However, this implies that, while moving around and changing their context, the users needs to discover and exploit the respective services (and applications) in each city.

From this analysis on the mobility domain, we can argue that mobility services are characterized by *gaps* and *potentialities*. Besides the huge amount of mobility services available up today, it is still missing for the users the possibility of getting *context-aware*, *accurate* and *personalized* travel solutions while moving around, without the need of discovering and using a multitude of different applications.

In this context, we claim that there is no need for yet-another-mobility-app. Our goal, instead, is to provide a solutions for *enhancing mobility services interoperability* through their run-time and context-aware discovery and composition, in order to exploit their potentialities and fill their gaps. This will allow users to get personalized solutions for their journey, from the planning phase till the journey execution. This can be done by leveraging together on the coverage offered by global services and on the data accuracy of local services, depending on the context and in a complete transparent way.

The challenges highlighted in this section motivate, among other things, our research on a design for adaptation approach of adaptive service-based systems allowing to overcome gaps and heterogeneity among services, independently of the domain they belong to. In particular, from our analysis of the mobility domain, it emerges that organizing and managing the mobility services, meeting travelers expectations and properly exploiting the available services, is still an increasingly complex task. This aspect has strongly supported our choice of focusing on the mobility domain and, particularly, on the design and implementation of a world-wide travel assistant as the running scenario of this dissertation.

7.2.2 ATLAS Platform

In this section, we present how ATLAS, our world-wide personalized travel assistant [29], has been developed on top of our design for adaptation framework. ATLAS consists both in (i) a demonstrator showing the system's models and its execution and evolution through automatic run-time adaptation, and (ii) a Telegram chat-bot, for the interaction with the users. Furthermore, ATLAS exploits *real-world mobility services* exposed as open APIs, which are wrapped as domain objects to be effectively part of the system. In this section, we introduce how ATLAS is organized as a service delivery platform. In section 7.2.3, instead, we show how adaptive service-based systems, such as the travel assistant, can be realized on top of it. From a technical perspective, the platform is organized in three main layers, as shown in Figure 7.3.

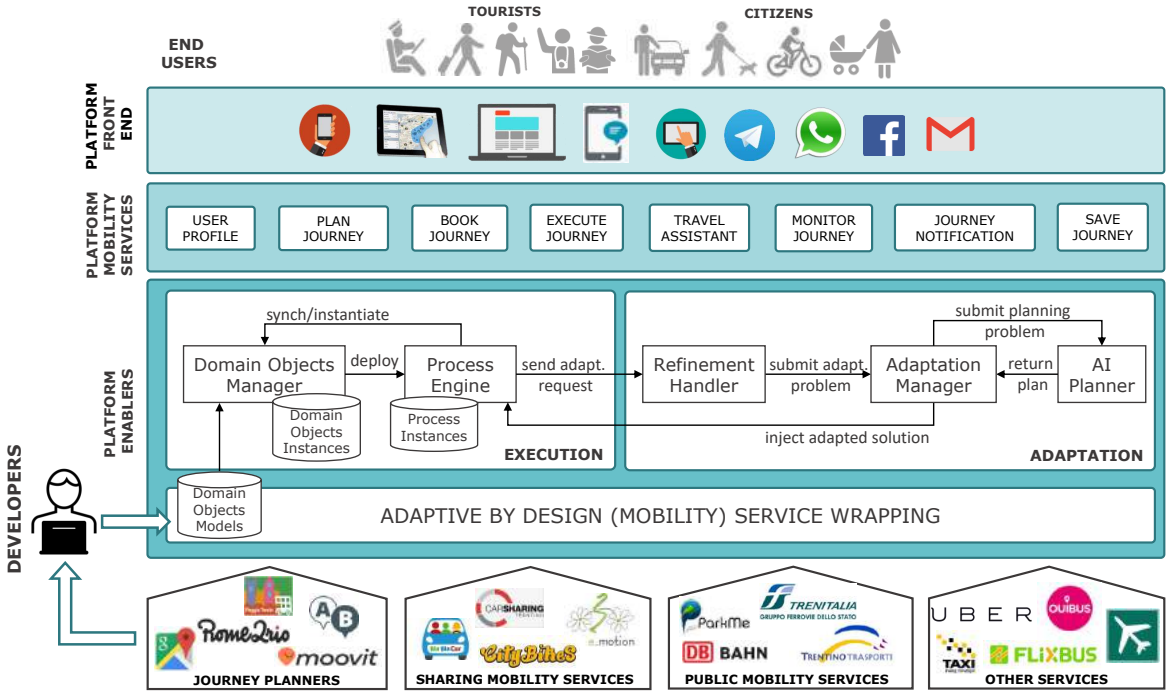


Figure 7.3: Domain Object-based Platform.

The **Enablers** leverage on our results on the adaptive by design wrapping of (mobility) services, based on the design for adaptation approach [27, 28] described in chapter 4. Developers can exploit and wrap up as domain objects the available services in the mobility domain. These ser-

vices can range from journey planners to public transport services, from shared mobility services to other mobility services not belonging to the previous categories. Besides the design of mobility services, enablers allow also for their run-time operation. All the details about the *execution* and *adaptation* enablers (see the Platform Enablers layer in Figure 7.3) have already been provided in chapter 5, section 5.2. In particular, we described (i) how the *execution enablers* perform the execution of the domain objects processes (i.e., core processes and fragments) during the operation of service-based applications, and how (ii) the *adaptation enablers* implement the adaptation mechanisms and strategies, described in section 5.1 and manage the adaptation needs of applications, arising at run-time.

The **Mobility Services** layer exposes the functionalities implemented or facilitated by the Enablers. These services can exploit and/or combine into value-added services the functionalities of the mobility services previously wrapped up and made available by the Enablers. In this layer, we can find services for the user profiling allowing to manage the user preferences. There are services for the planning, booking and execution of journeys, as well as services allowing users to monitor, save and get notified about the scheduled journeys. The key idea is that the platform is open to continuous extensions with new services, wrapped as domain objects. Their functionalities can thus be exploited in a transparent way to provide value-added services to the end-users.

All the platform mobility services can be eventually provided to final users through a range of multi-channels front-end applications that constitute the **Front-end** layer. These can be mobile or desktop applications, and they can be independent or rely on existing services, such as chat-bots (e.g., Telegram chat-bot).

7.2.3 ATLAS Implementation

As already introduced, ATLAS consists of both a demonstrator and a Telegram chat-bot application. In particular, the demonstrator is based on a process-engine implemented in-house. This handles the *multiple-instance processes management*, the dynamic *correlation among processes* and the *abstract activity management*, which are needed requirements for the appli-

cation of our techniques. The demonstrator also implements the enablers described in section 5.2 (e.g., Adaptation Manager, Domain Objects Manager). In this section, with the ATLAS platform in mind, we give more details on the ATLAS travel assistant , to show how different applications (i.e., the ATLAS telegram chat-bot) can be realized, and further executed, on top of the platform.

To realize a world-wide travel assistant able to provide to the users the proper mobility services in the specific context(s) of their journeys, we selected real-world mobility services exposed as open APIs. We identified their behavior and functionalities and their input and output data. Finally, we wrapped them up as domain objects to be stored in the platform knowledge base. For instance, we wrapped *Rome2Rio* and *Google Transit* as global journey planners. However, to overcome the limitations of global planners in terms of accuracy, local planners are needed. In our platform, for instance, we wrapped *ViaggiaTrento*, as local planner. It can be exploited for those journey located in Trento, which can also be part of a wider inter-modal travel solution provided by a global planner, but for which the global planner does not give enough or accurate information. Indeed, for long distance travel solutions, probably touching several cities, global planners are not as accurate as local planners of those cities. Combining the geographical coverage of global planners with the accuracy of local planners is a concrete example of services interoperability promoted by our platform. Going on, other open mobility services have been considered. We mention *Travel for London* [16] as planner for the city of London, *BlaBlaCar* as ridesharing service, *CityBikes* as bike sharing service applying to about 400 cities, *Trentino Trasporti* [17] for the public transportation in the Trentino region, as examples. Being defined as domain objects, all these services can now be executed, automatically composed and adapted by the enablers of the platform.

At the *Mobility Services* platform level, instead, we can find the *Travel Assistant*. It has been defined as a value-added service leveraging on the services available in the system. In particular, its main features are the following:

[16] `https://api.tfl.gov.uk/` [17] `http://www.ttesercizio.it/`

1. given a user planning request, it is able to decide between a *local* or a *global* planning service to better handle the request, by analyzing the source and destination points entered by the user.

2. Given the planners responses, it defines the better way to show this responses to the user (e.g., a list of travel alternatives, a warning message). Different data patterns are provided on the basis of different data format (e.g., XML, JSON) and relevant information. Moreover, the application allows also the data in the service responses to be filtered and ordered on the basis of the user preferences (e.g., price, distance, travel time). Patterns are applied for each used mobility services and not only for journey planners.

3. Once the user selects a specific solution, the travel assistant is able to identify the transport means in the legs making the entire solution. With the information that it gets, it goes vertically to find the proper service(s) to use (e.g., the ones of the specific transport companies), if existing in the system. Moreover, these services can refer to different aspects of the organization of a journey, such as the presentation of timetables, the booking of tickets, the notification of delays, etc. In this way, the application can incrementally provide to the user specific functionalities and context-aware information for his/her journey, without requiring users to look for and interact with different and independent applications.

We will see concrete execution examples further on in this section. We emphasize here that the more (mobility) services are wrapped up and stored in the system's knowledge base, the more responsive and accurate the travel assistant will be.

Finally, among the multi-channel front-ends that can be defined on top of the platform, we realized the travel assistant as a Telegram chat-bot, exploiting the open API provided by Telegram. The same travel assistant might be furnished via a different front-end, too.

A graphical representation of (a portion of) the travel assistant system, as the domain objects hierarchy arising from the wrapping of mobility services and the travel assistant design, is given in chapter 4, in Figure 4.3.

In the rest of this section, instead, we focus on the execution of the travel assistant, and on its interfaces, both that of the demonstrator and that of the Telegram chat-bot.

ATLAS Demonstrator Interface. The execution of ATLAS is triggered by the users interacting with the ATLAS Telegram chat-bot installed on their smartphone. The use of the chat-bot allows users to interact with services that are dynamically selected and composed by the framework, based on their requests, in a completely transparent way. For this reason, we decided to realize also a demonstrator showing the background execution of the travel assistant performed through the dynamic specialization of processes, via the abstract activities refinement mechanism. In this paragraph, we show the interface of the ATLAS demonstrator, its main windows and how it works. Examples of running scenarios as shown on the Telegram chat-bot users interface, instead, are given in the next paragraph.

First of all, the ATLAS demonstrator is made by two main tabs, namely the *Domain Objects Models* tab, in Figure 7.4, and the *Runtime Execution* tab, in Figure 7.5.

The Domain Objects Models tab is devoted to show the models of the adaptive by design services that have been wrapped as domain objects and stored in the system knowledge base. On this tab, a user can select a specific service from a provided list (Figure 7.4 left side), that is, its corresponding domain object, and inspect the models of its core ingredients, namely the *Core Process*, the *Provided Fragments*, the *Domain Knowledge* and the *Domain Object Definition*. They are all organized on a tab view (see the bottom side of Figure 7.4), among which the user can easily move from a construct to another. In particular:

- the *Core Process* tab shows the APFL model of the domain object's core process, with the sequence of all its activities (i.e., input, output, concrete and abstract activities), the specific used constructs (i.e., while loops, switch, if), the annotations labeling the activities (i.e., preconditions, effects and goals) and the transitions among them.

- the *Provided Fragments* tab, which is exactly that shown in Figure 7.4, gives the list of the fragments exposed by the selected domain

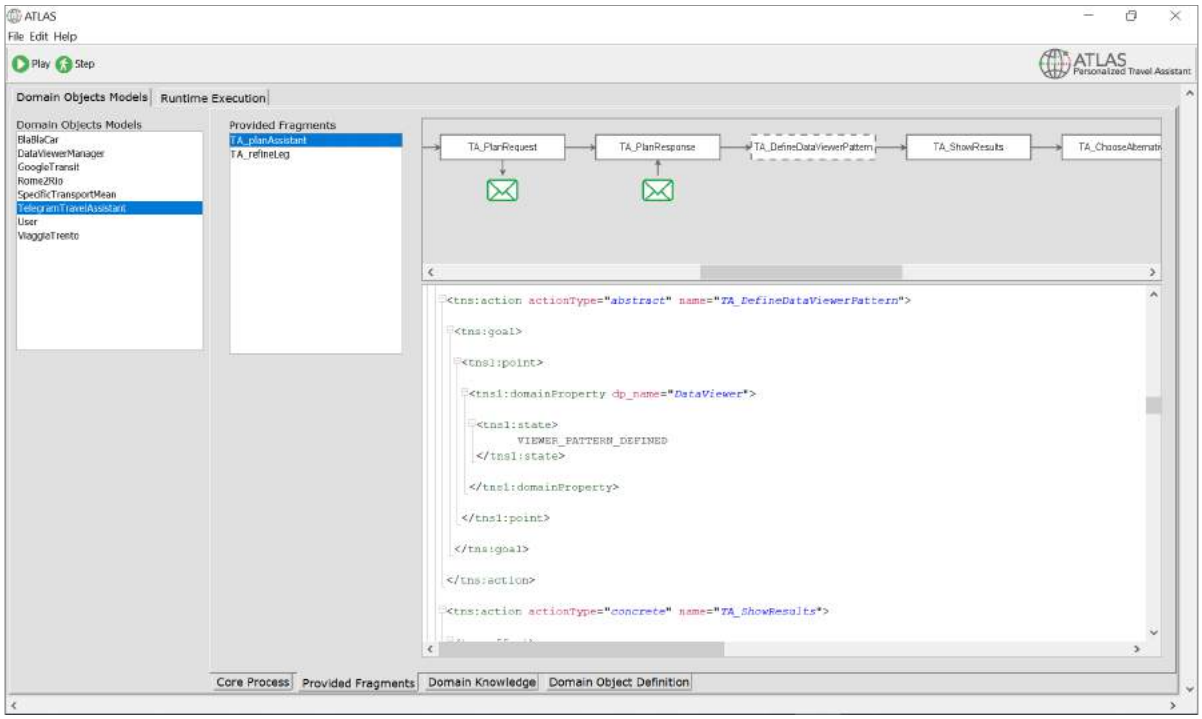


Figure 7.4: ATLAS Demonstrator – Models view.

object. For each fragment it shows both a graphical and a textual representation. While the textual model is XML-based, in the graphical representation of fragments, we use different ways to shape the different kinds of activities. In particular, *output* and *input* activities are depicted with outgoing and ongoing messages, respectively. Indeed, these activities model the I/O communications among the fragment and its relative core process. *Concrete* activities are depicted with solid lines, while *abstract* activities are depicted with dotted lines.

- the *Domain Knowledge* tab reports the list of domain properties belonging to both the *internal* and *external* knowledge of each domain object. This tab is structured exactly as the Provide Fragments tab. The only differences are that (i) the textual description gives the XML representation of the state transition system corresponding to the specific domain property (i.e., states, transitions, events), and (ii) the graphical representation shapes a state transition system.

- the *Domain Objects Definition* tab, eventually, reports the XML-

based model of the selected domain object, where all its core ingredients and attributes (e.g., state variables) are specified.

At this point, we can describe the Runtime Execution tab that is, instead, devoted to the execution of service-based systems (i.e., the travel assistant) developed within our framework. Thinking at the travel assis-

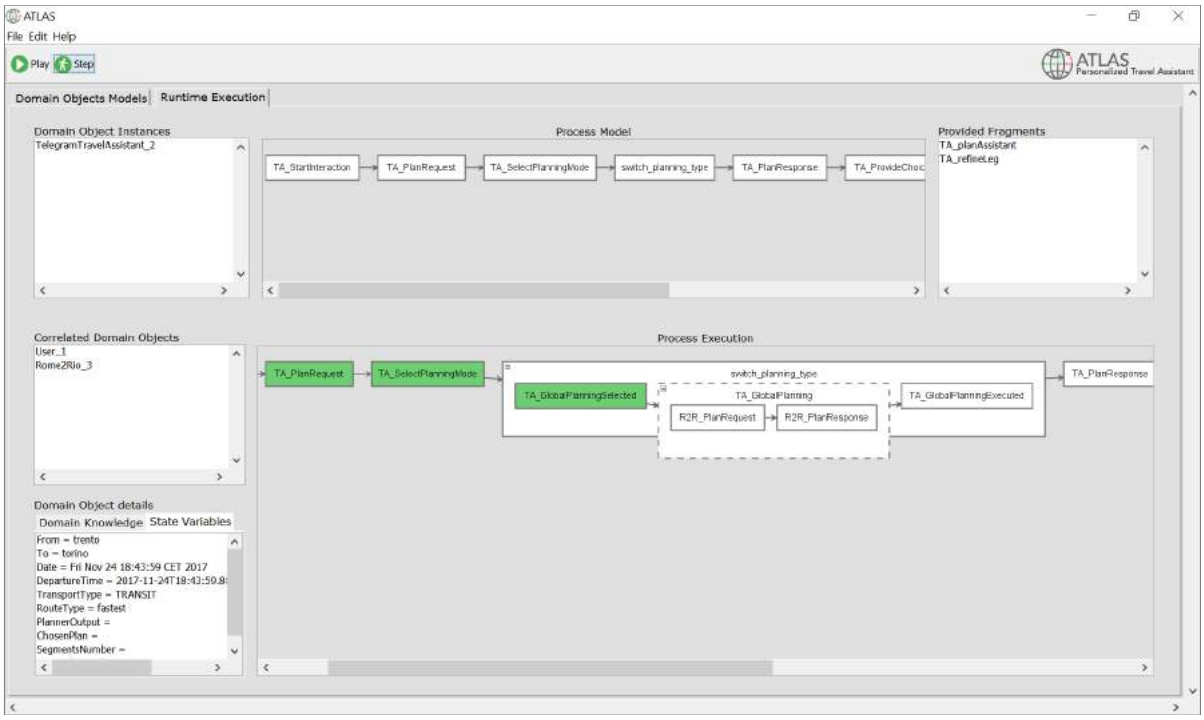


Figure 7.5: ATLAS Demonstrator – Execution view.

tant as running system, the ATLAS demonstrator allows the starting of the travel assistant, with the button *Play* as well as its step-by-step execution, with the button *Step*, on the top-side toolbar. Once started, it waits for a user request from an instance of the Telegram chat-bot, in order to subsequently show its dynamic process execution. The Runtime Execution tab is organized in such a way that:

- for each running domain object instance in the system, it is possible to follow its execution and evolution;
- for the specific user of the travel assistant, it is possible to see how his/her specific travel assistant instance evolves through the dynamic

establishment of co-relations among domain object instances that dynamically interoperate by exchanging fragments among them. In particular, for each user, different domain objects will be instantiated, the ones corresponding to the context-aware services providing the fragments needed to accomplish their specific needs.

Indeed, during the run-time execution of the system, the domain objects' hierarchy, defined by the *soft dependencies* among domain objects, as emerging at design time (see Figure 4.3 as an example), evolves thanks to the *dynamic inter-operation* among domain objects and the *domain knowledge extension* (see section 5.3 and section 5.3.1, respectively), to which each domain object might be subject while executing.

Going back to the Runtime Execution tab of the demonstrator, on its top left corner it provides the list of the *Domain Objects Instances* currently running in the system. If an instance is selected, the whole interface is updated accordingly. More precisely:

- the *Correlated Domain Objects* area shows all the domain objects instances to which the selected domain object has a co-relation. A co-relation between two domain objects is established when a domain object injects in its core process a fragment provided by another domain object, thus transforming their soft dependency into a strong dependency;
- the *Domain Objects Details* tab view shows the domain properties in the domain knowledge of the current domain object, by giving the state in which they are (in the *Domain Knowledge* tab). The list of all the variables making the internal state of the domain object, and its evolution (in the *State Variables* tab);
- the *Provided Fragments* area (top right corner in Figure 7.5) lists the fragments offered by the current domain object;
- the *Process Model* area loads the model of the current domain object core process;
- the *Process Execution* area, instead, loads the running domain object's core process instance and it shows its dynamic execution via the

incremental composition of services. The process in execution is displayed by green-coloring the executed activities and by highlighting the (chain of) refinements of abstract activities, with the corresponding injection of fragments compositions, as shown in Figure 7.5.

In chapter 5, we provided an overview on the dynamic adaptation of fragment-based and context-aware business processes [25, 120], in order to comprehensively define the adaptation mechanisms and strategies exploited by our approach. Then, we specified that in the current version of the approach we deal, in particular, with the abstract activity refinement mechanism. Moreover, we also described the fragments composition approach based on AI planning [47]. Essentially, for the automatic resolution of an adaptation problem, according to [47], this is transformed into a *AI planning problem*, so that planning techniques can be used to solve it.

In order to give more details about the refinements of abstract activities, when these are performed, we realized a window specifically devoted to the application of the refinement mechanism, allowing the inspection of the automatic resolution of an adaptation problem. By clicking on an abstract activity of a running process in the Process Execution area, the window in Figure 7.6 is opened. On its first tab, the *Process/Fragment Activity*, it shows details about the abstract activity currently under refinement, such as its goal. Then, the window is made by the following other tabs.

In the *Activity Specialization Problem* tab, in Figure 7.7, the *adaptation problem* is specified by a set of *fragments* and a set of (involved) domain properties belonging to the external knowledge of the domain object that is currently executing the refinement.

On the *Planning Domain* tab, in Figure 7.8, the planning domain is reported after being translated into a `*.smv` file specification by the *Composer*, in order to be given as input to the *AI Planner* (see the component diagram in Figure 7.1).

Eventually, the *Activity Specialization Result* tab, in Figure 7.9, reports the (composition of) fragments returned by the Planner as result of the abstract activity refinement process. It is injected in place of the abstract activity, and its execution allows its goal to be reached.

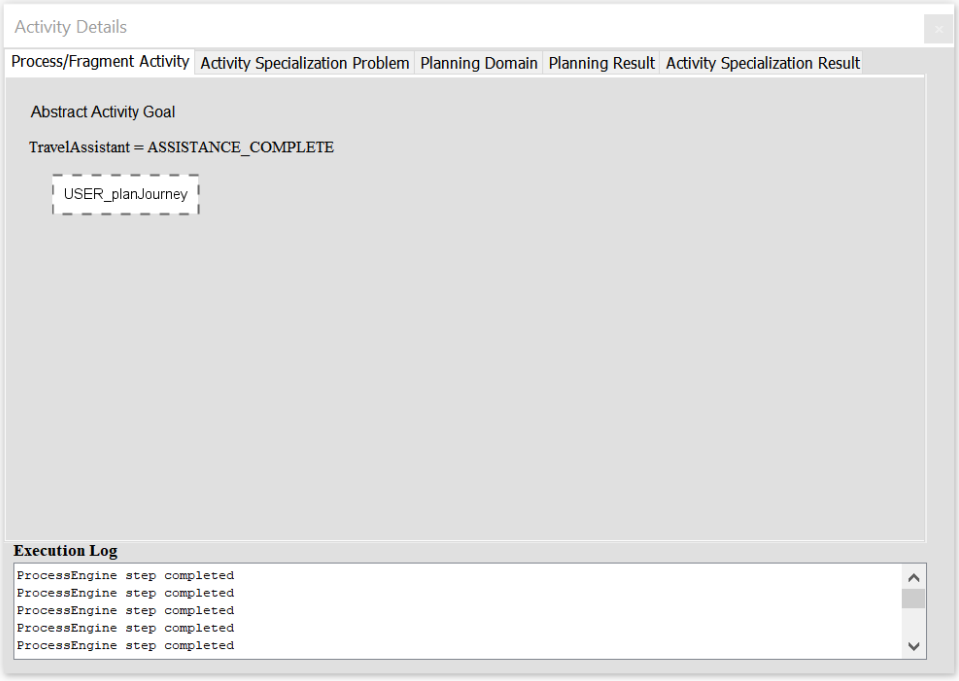


Figure 7.6: Abstract Activity Refinement Window.

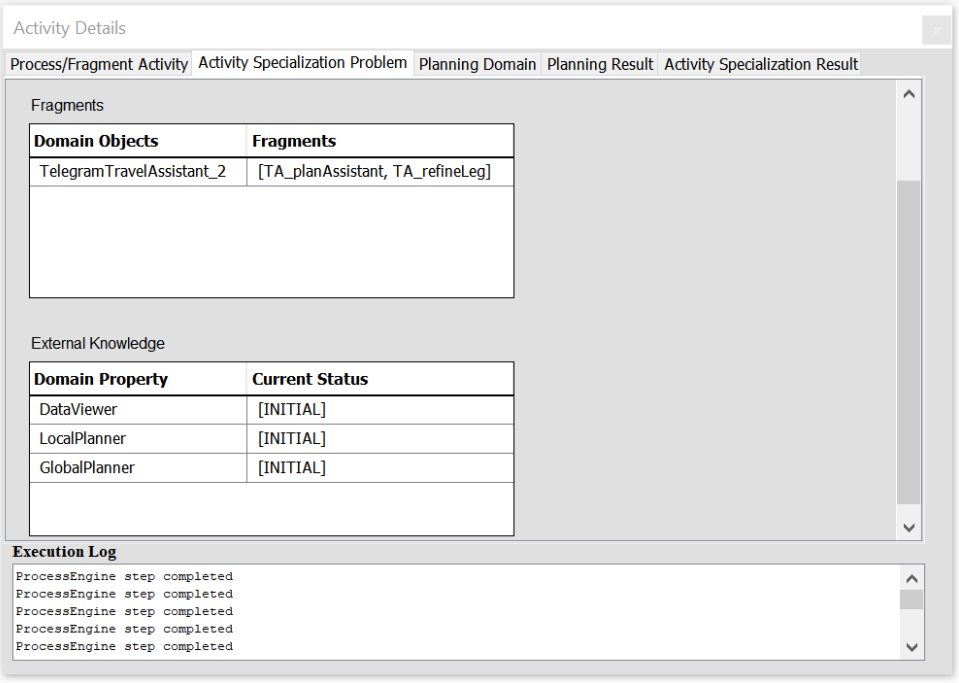


Figure 7.7: Process Specialization Problem Tab.

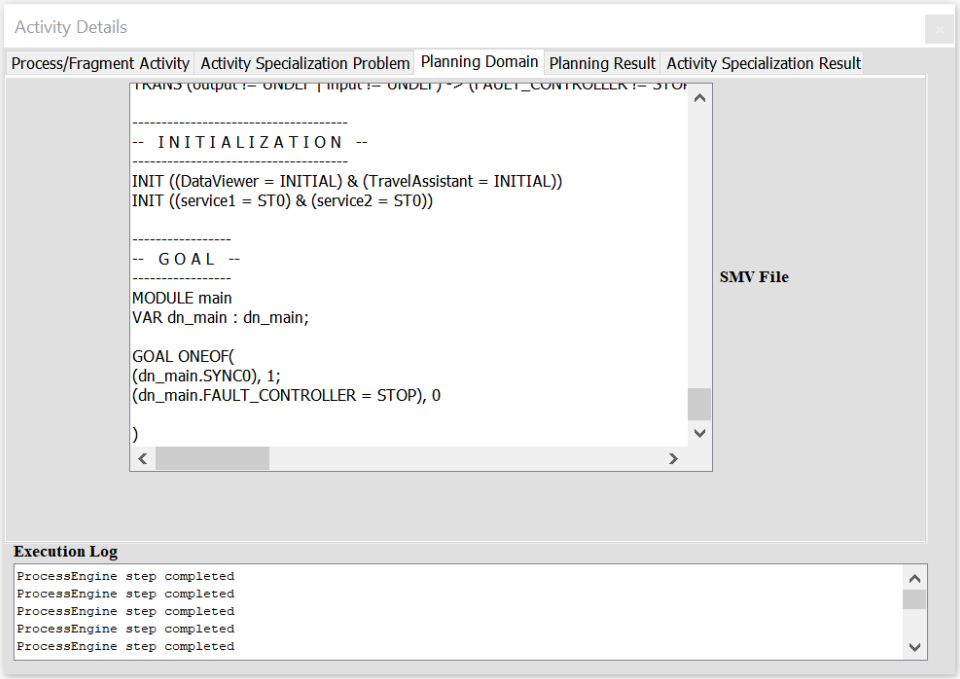


Figure 7.8: Planning Domain Tab.

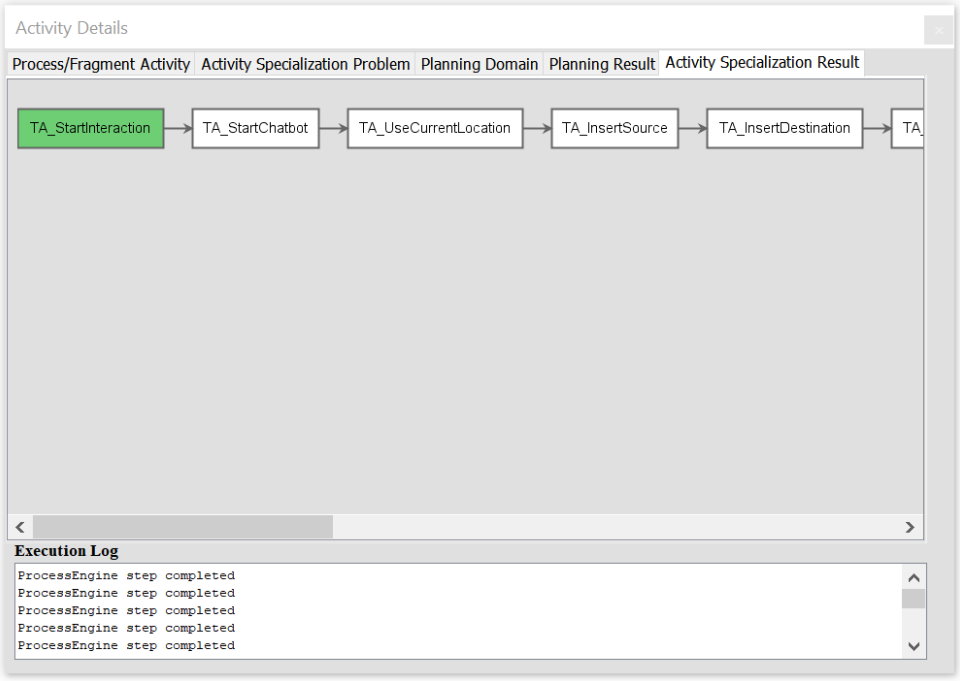


Figure 7.9: Process Specialization Tab.

ATLAS Chat-bot Execution. In this paragraph, we show the execution of two different scenarios, representing two relevant use cases of ATLAS, from the perspective of the users exploiting the travel assistant in their daily life. Indeed, as we said before, one of the concrete example of services interoperability promoted by the ATLAS demonstrator consists in combining the geographical coverage of global planners with the accuracy of local planners. In other words, while looking for context-aware services to be dynamically exploited, after a journey planning request from a user, the travel assistant is able to decide between a *local* or a *global* planning service to better handle the request. For this reason, we show how ATLAS runs in the Telegram chat-bot interface, both in a situation of a local journey organization, and in a situation of a global journey organization. The *chain of incremental refinements* that is dynamically set up from the execution of the following scenarios, is similar to that given in Figure 5.4 of section 5.3.

Examples of ATLAS use cases, whose execution can be observed also within the demonstrator, are the following.

Local journey organization use case. Sara lives in Trento, Italy, and she wants to find her way to reach the Christmas markets located in Piazza Fiera, not so far from her departure place, which is the public swimming pool in via Fogazzaro. In Figure 7.10, we show the relevant screenshots of the ATLAS chat-bot running on her smartphone's display, as the user interface showing both how Sara can interact with ATLAS and how the information are displayed. The chat-bot allows Sara to enter her departure and destination points (see the screenshot on the left side in Figure 7.10). Being both places located in the same city, Trento, the travel assistant understand that a local planning would be more appropriate for the specific user request. Thus, it dynamically finds and selects the *Viaggia Trento* journey planner, a local planner for the city of Trento exposing a fragment for the journey planning execution. The journey planner's response if further handled and parsed by the `Data Viewer` domain object devoted to the visualization of information on the chat-bot, which has been previously described in chapter 4. The result is shown to Sara as in the central screenshot in Figure 7.10. Since in the request she opted for a healthy solu-

Figure 7.10: Screenshots of the ATLAS chat-bot – Local Journey Organization.

tion, the Viaggia Trento journey planner replies with the healthier solution available, which is the one provided by a bike-sharing service, whose racks are close to both her source and destination places. At this point, Sara would like to know if there are available bikes to be used. To this aim, the travel assistant continues with its execution and it identifies the bike-sharing service available in Trento, namely *e-motion*. Since this service has been wrapped as domain object and, thus, it is available in the system, the travel assistant selects its fragment whose execution allows the system to get information about the available bikes at the closest bike-sharing racks. Again, the result is shown by the chat-bot on the smartphone of Sara, as in the right-side screenshot in Figure 7.10. Sara knows, now, that 3 bikes over 11 are still available at the rack close to the swimming pool (first element in the result list). The e-motion bike-sharing service does not allow for the booking of bikes, so that the execution of ATLAS stops here. In summary, for her specific request, Sara transparently interacted with different mobility services that have been dynamically selected and composed for her specific request, taking into account her specific context. This is a typical scenario of an execution of ATLAS in the *local* case.

Global journey organization use case. The second execution example, instead, refers to Paolo. He must organize his working journey from Trento to Torino and he also is a user of ATLAS. The relevant screenshots of his journey organization are reported in Figure 7.11.

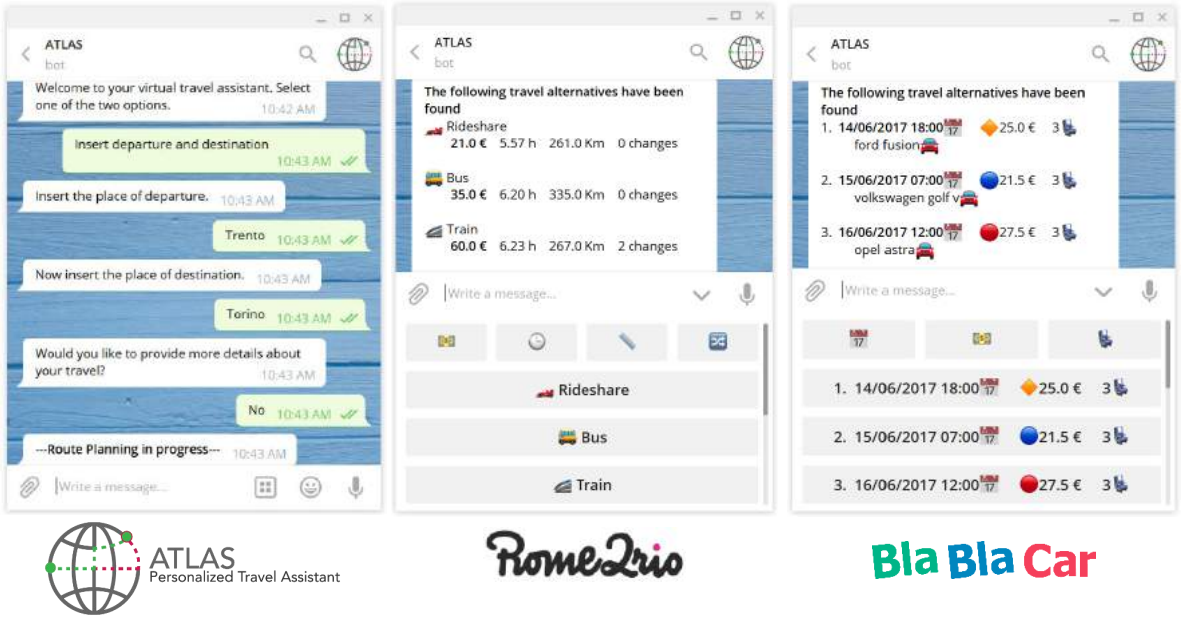


Figure 7.11: Screenshots of the ATLAS chat-bot – Global Journey Organization.

He starts by entering his departure and destination places, as in the screenshot on the left side. Being Trento and Torino two different cities, the travel assistant opts for a global planning solution. It finds and selects a fragment for planning a journey exposed by the *Rome2Rio* global journey planner. The found travel alternatives are shown to Paolo as in the central screenshot in Figure 7.11. Different alternatives are available (e.g., rideshare, bus, train, etc). In this example, we can see how ATLAS displays relevant details for each solution, such as the cost, the duration, the distance and the number of changes. Moreover, it also allows these solution to be differently ordered (e.g., by price, by cost, etc), by using the buttons on top of the chat-bot keyboard. At this point, Paolo selects the first alternative, namely the rideshare one, which is also the less expensive. In particular, the ride-sharing solution suggested by Rome2Rio is provided by the *BlaBlaCar* ride-sharing service. In order to get more information

about the BlaBlaCar travel alternatives and their details (e.g., departure time, available seats, driver contact, etc), the travel assistant continues its execution. It dynamically finds and selects the fragment exposed by the BlaBlaCar service, whose corresponding domain object exists in the system knowledge base. Its execution provides detailed information that are further shown to Paolo, as in the right-side screenshot in Figure 7.11. We highlight here that, to use the BlaBlaCar service, it is not required that Paolo has the BlaBlaCar mobile application installed on his smartphone. Instead, to continue with the booking of the ride-share solution, it is required that he is subscribed to the service. To sum up, also Paolo has been able to plan his trip to Torino, by interacting with different mobility services, specifically selected for his initial request.

In conclusion, these execution examples exhibit two important aspects of our approach. Firstly, they show its *bottom-up* nature, where mobility services functionalities go through the domain objects hierarchy (refer to Figure 4.3) till the user process where they are executed. Secondly, this happens in a completely transparent way for the user that interacts with only one application, the ATLAS chat-bot.

7.2.4 ATLAS Evaluation

To evaluate ATLAS, both in terms of *effectiveness* and *efficiency*, we have run a set of experiments. In this section we describe their design and we discuss the collected results. The tests are done on real-world problem that were generated by randomly choosing an origin and a destination points while running ATLAS. The specification of ATLAS used to evaluate the platform contains 14 domain object models, 17 fragment models and 12 types of domain properties. Domain properties are high-level representations of the domain concepts, and they are used to evaluate the conditions under which each fragment, provided by a domain object, can be exploited. We ran ATLAS using a dual-core CPU running at 2.7GHz, with 8Gb memory. To show its *feasibility*, we evaluate the following aspects:

- How long it takes to wrap up real services as domain objects;

- How much automatic refinement (service selection and composition) affects the execution of the travel assistant.

To answer to the first point, and based on our experience acquired during the development of ATLAS, we can argue the following. To wrap a real service as a domain object, the developer needs (i) to master the domain objects modeling notation and (ii) to understand the service behavior, its functionalities, its input/output data format and how to query it. Wrapping time clearly changes between experienced and non-expert developers. From our analysis, it ranges from 4 to 6 hours, considering average complex services. Moreover, it is also relevant to claim that this activity is done *una tantum*: after its wrapping, the service is seamlessly part of the framework and exploited for automatic composition and refinement.

To answer to the second point, we collected both the adaptation and mobility services execution statistics, to understand how long they take, on average, to be executed. To evaluate the automatic refinement, we carried out an experiment in which we considered 10 runs of ATLAS handling various end-users' requests. We collected adaptation data such as the number of adaptation cases, their complexity and the time required to generate adaptation solutions. For each run, more than 150 refinement cases were generated. Figure 7.12 left side shows the distribution of problem complexity considering the 10 runs. The complexity of an adaptation problem is calculated as the *total amount of transitions* in the state transition systems representations of the domain properties and fragments present in the problem. For simplicity, in the graph we aggregated the problem complexities in ranges of 20. The majority of the problems have a complexity in-between 0 and 19 transitions. A significant number of them have a complexity in-between 40 and 59 transitions, while the most complex problems have a complexity ranging from 80 to 100 transitions. Notice that the occurrence of complex problems is relatively rare (in this real-world battery of tests). Figure 7.12 right side shows the percentage of refinement problems solved within a certain time. We can see that, for all the runs, 93% of problems are solved within 0.2 seconds. Only 3% of the problems require more than 0.5 seconds to be solved, and the worst case is anyhow below 1.5 seconds.

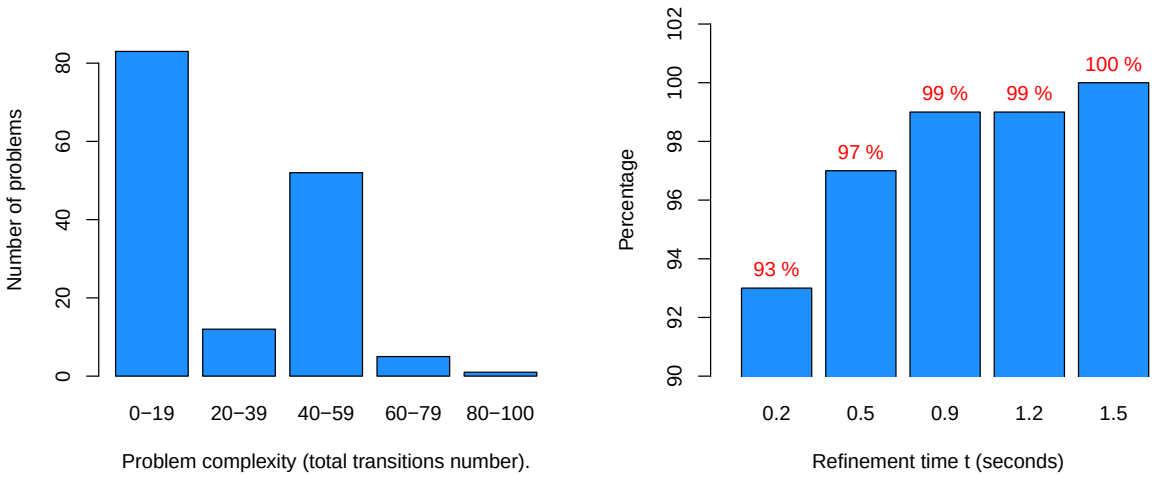
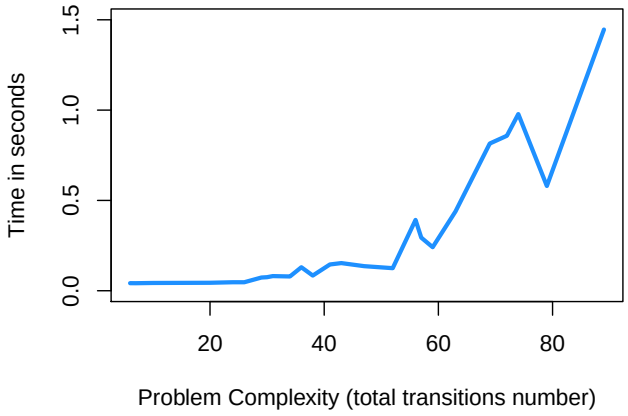


Figure 7.12: (Left side) Distribution of Problems Complexity. (Right side) Percentage of problems solved within time t.

To measure how much automatic refinement influences the execution of ATLAS, we compared the data about the time required for adaptation with the response time of real-world services wrapped in ATLAS. Figure 7.13 left side relates the (average) time required to solve a composition problem to the problem complexity. The average time is computed considering in the 10 runs all the refinement problems having the same complexity. As expected, problems with higher number of transitions (and hence the most complex planning domain) take more planning time than problem with less complexity. Figure 7.13 right side, instead, relates to the (average) response time of (a subset of) real mobility services, which are part of ATLAS. We can notice that, in the worst case, the adaptation requires a time close to 1.5 seconds, while the services response time ranges from 0.23 to 3.20 seconds. Moreover, the adaptation takes more time for the most complex problems that, however, are the less frequent to be executed. We can argue that the automatic refinement responsiveness is equivalent to that of mobility services. In conclusion, these results demonstrate the *effectiveness* and the *efficiency* of our approach when applied to a real-world complex scenario.



Services (Avg) Response Time	
Service Name	**Resp. Time**
Bla Bla Car	0.78 secs
City Bikes	0.23 secs
Google Transit	0.65 secs
Rome 2Rio	1.20 secs
Travel for London	3.20 secs
Viaggia Trento	0.77 secs

Figure 7.13: (Left side) Trend of the Adaptation Time. (Right side) (Average) Services Execution Time.

7.3 DeMOCAS: Domain Objects for Service-based Collective Adaptive Systems

In this section, we introduce DeMOCAS – *Domain Objects for Service-based Collective Adaptive Systems*. It is a simulator showing how the design for adaptation approach, extended to deal with *ensembles* made by collaborative and interacting entities (i.e., domain objects), *supports the development and execution of collective adaptive systems operating in dynamic environments*. We further discuss the effectiveness of the approach when applied to smart cities applications like the "Urban Mobility System". Indeed, for the implementation and evaluation of DeMOCAS, we designed the UMS in terms of domain objects and we simulated its execution through the dynamic formation of ensembles and the application of collective adaptations, when issues affecting an ensembles or the whole system arise.

Differently than what we did for the design of the travel assistant in ATLAS, in DeMOCAS we do not deal with real services. However, even if we defined domain objects representing non-real world mobility services, we designed them in a realistic manner, so that to realistically simulate the UMS behavior, as it is in a real city. For its execution, DeMOCAS lever-

ages on both the advanced techniques for dynamic and incremental service composition [128] (see chapter 5) and the collective adaptation algorithm [30] (see chapter 6), to effectively deal with changes occurring at different levels in the system.

While in chapter 6 we evaluated the collective adaptation algorithm in terms of its feasibility and scalability when managing real-sized applications, we exploit DeMOCAS to further discuss and verify the usability and scalability of the exploited adaptation mechanisms. Indeed, since adaptation is relevant for the execution of systems, by firmly affecting it, we intend to prove that its application does not degrade the performance of systems.

The rest of the section is organized as follow: in section 7.3.1 we deeply describe the UMS scenario, with examples about its execution and collective adaptation. In section 7.3.2 we provide an overview on the design of the UMS in terms of domain objects. Section 7.3.3 is devoted to describe the implementation of DeMOCAS, while in section 7.3.4 we provide evaluation results as obtained by running DeMOCAS.

7.3.1 Evaluation Scenario: the Urban Mobility System

In this section, we recall and further detail the Urban Mobility System scenario, previously described in chapter 6, which has been modeled and can be executed within DeMOCAS. We describe an overview of the system, together with examples of its execution, especially in the case in which a collective adaptation must be performed. As last, we focus on a portion of the entire system, the one related to the Flexibus transportation mean, since even though it is quite simple, it provides sufficient complexity to represent the concepts of *ensembles*, *roles* and collective behavior. A (partial) overview on the adaptive by design model of the UMS, in terms of *domain objects* and *domain properties*, is further given in section 7.3.2.

The UMS scenario is characterized by the active involvement of the citizens in the city life. They proactively contribute, by offering their capabilities, expertise and resources that may be deployed in different processes at the city level, for the benefit of the whole community. At the same time, supporting citizens mobility is a priority for municipalities. Indeed, the

goal of this scenario consists in exploiting in a synergistic manner the heterogeneous city transport services, while providing accurate, real-time, and customized mobility services to citizens. The system helps citizens to plan and execute a journey as well as to deal with context changes that may affect a journey with the introduction of alternatives or different planning solutions. In particular, the system provides *multi-modal*, *collaborative* and *customized* tasks.

In the following we briefly report the UMS characterization. The UMS integrates three means of transport: regular bus, flexible bus (*Flexibus*) and car pooling. As known, regular bus services deal with predefined routes and timetables supported by a number of buses. Flexibus, instead, is a modern transportation service, that combines the features of taxi and regular bus services. A Flexibus system manages on-demand routes defined as a network of stops (pickup points) and provides to passengers transportation between any of these points. At last, car pooling provides integration between independent drivers and passengers. The goal of the UMS is to provide the integration of these services to the passengers and to support them to collaboratively addressing the context changes affecting their journeys. Different actors are involved in such a scenario: there are citizens representing the main users of the whole system, acting as passengers of the different transportation means. Then there is a manager (aka company, e.g., the Flexibus Company) for each transport service that is devoted to the management of the service itself and that acts as a mediators between passengers, employees of the company, routes covered by the means, third party services on which it relies on, and so on. Drivers are also actively involved by dynamically interacting both with the system and with the passengers for giving and receiving updates on their assigned routes. At last, the UMS represents the supervisor actor that provides integration between all the involved actors (see Figure 7.14).

In the rest of this section, we consider only the Flexibus transportation mean. Although simple, this portion of the UMS scenario is sufficient to show the relationships between the involved entities, their dynamic interconnections and cooperation. Within the current scenario we can distinguish the following set of entities:

- *User*. The user entity models a real citizen that uses the UMS to plan her journey. The UMS will support her during the journey organization (e.g., journey planning, tickets booking), through its execution (e.g., reaching flexibus pickup points, monitoring transportation means).

- *Flexibus Employee*. The Flexibus Employee entity models the system's users representing the employees of the flexibus company. They can play different roles; in this version of the scenario we defined the role of the flexibus driver.

- *Urban Mobility System*. This entity represents the mediator between system's users and transportation means (e.g., flexibus) and it provides their integration. It exposes to the passengers the available tools for planning multi-modal trips, and it allows passengers to establish connections with the entities related to her plan, in order to interact with them for defining, executing and finalizing the plan. It also manages integrated notification and support system to pursue collective adaptation objectives.

- *Trip Organization*. This entity concretely organizes a multi-modal trip on the basis of the plan chosen by the passenger between a set of provided alternatives. When the plan is defined, it interacts with the transportation companies involved in the plan to accomplish all the organizational tasks (e.g., booking, ticket payments, seat reservation).

- *Multi-modal Planner*. This entity is devoted to provide multi-modal plans based on users requirements.

- *Flexibus Company*. This entity implements the logic of flexibuses organization, and represents the interface with the service users. It collects booking requests from the passengers, manages the creation of routes and it assigns drivers to them.

- *Route Passenger*. This entity represents the passenger of a flexibus route. It specifies the behavior that a user must implement when she becomes passenger of a flexibus.

- *Flexibus Driver*. This entity has the goal to perform his assigned route by covering all the pickup points composing it, while respecting the passengers needs, such as time requirements. He is in touch with the Flexibus Company (e.g., to get the route), with the Route Manager (e.g., to get routes info, such as the list of pickup points and the list of passengers, the route updates) and with the passengers (e.g., to communicate with them in case of delay).
- *Route Manager*. It has the goal to manage specific routes, each composed by a set of pickup points, a set of passengers and a Flexibus Driver.

Moreover, for adaptation purposes, each of these entities is able to interact with the other entities which it has relations with, to *notify*, *solve* or *manage* problems, as well as to adapt their own processes in order to apply adaptation solutions.

UMS Execution. Given the involved entities, we briefly describe how they can interact and which relationships can potentially be established among them, during the execution of the UMS.

First of all a user interacts with the system to plan a journey. The system provides the right functionalities to define multi-modal alternatives and to organize trips. To plan trips, the *UMS* relies on functionalities exposed by entities in the system offering planning functionalities, if any; in this scenario this is the case of the *Multi-Modal Planner* (MMP) that is able to plan trips by considering all the transportation means covering the area in which the trip has to be executed. Moreover, the MMP provides all the solutions applying to the user request. Eventually, the solutions are forwarded to the user that chooses her preferred solution and asks the UMS to proceed with the trip organization. To accomplish this task, the UMS relies on functionalities offered by other entities in the system dealing with the management of journeys, spanning from the inizialization phase to the post execution phase. The *Trip Organization* entity receives the multi-modal alternative selected by the user. On the basis of the transport means involved in it, the Trip Organization dynamically establish relations with the entities providing the needed functionalities (e.g., booking, ticket

payments, registration and login) and it forwards to the user the knowledge to interact with them.

For instance, we consider a simple example where the solution involves only the flexibus transport mean. The *Flexibus Company* is the entity acting as interface of the whole flexibus system. Thus, through the Trip Organization, the user establishes a new relation with the Flexibus Company which allows her to login or register to the system, to book for the trip and pay the ticket on-line if possible. When all the details of the trip are defined and approved, the user becomes a *Route Passenger*, since she is assigned to a specific flexibus route. This means that she will interact with the Route Passenger entity, which provides all the facilities to check-in at the pickup point, check-in on the flexibus, communicate with the driver and the other passengers if needed, and so on. The *Route Manager* entity, instead, is devoted to the creation and management of flexibus routes. It provides functionalities both to the Flexibus Company, in order to create new routes and pickup points on the basis of the users bookings, and to the Route Passenger in order to check-in on routes. At last, the employees of the flexibus company (*Flexibus Driver*) are also users of the system. They interact with the Flexibus Company entity to be assigned to specific routes and, for each route they must travel, they interact with the Route Manager to update it and to be up to date about the route execution. We refer to section 7.3.2 for an overview on the domain objects modeling the just mentioned entities and the soft dependencies among them.

UMS Ensembles. As introduced in chapter 6, to allow system entities to collectively adapt, dealing with adaptation needs that can be raised both by the environment and by the entities themselves, the domain objects model has been extended with specific constructs. A key concept are *ensembles*, that are modeled over the dynamic network of domain objects, as groups of domain objects playing different roles. Although autonomous in their execution, domain objects belonging to the same ensemble share common goals and might need to collectively handle run-time adaptation problems. Indeed, in dynamic contexts such as that of the UMS, isolated entity self-adaptation is not effective. We can imagine what happens if a passenger books a trip with a flexibus and then silently decides not to

travel. It is likely to cause unnecessary delay for the route (e.g. the bus will have a redundant stop) and raise the cost of the trip for the remaining passengers. Even more serious consequences arise if a bus gets damaged: isolated adaptation by the bus driver could totally break the passengers travel plans.

In adaptive systems with collective behavior approaches for collective adaptation are therefore needed to allow (i) multiple entities to collectively adapt with (ii) negotiations to decide which collective changes are best. Moreover, a relevant challenge refers to which parts of the system should be engaged in an adaptation. Indeed, solutions for the same problem may be generated at different levels.

Example. *For instance, a passengers delay may be resolved in the scope of a flexibus route, by re-planning the route, or in the wider scope of the flexibus company, with the engagement of other routes, or even in the scope of the whole UMS, with the engagement of other means of transportation such as a car pool.*

The *hierarchical nature* of an UMS opens up all these alternative options. Within our scenario, we can identify several hierarchical levels of abstraction that operate at different scales in time and space, as in Figure 7.14. A flexibus route combines passengers with a driver, a Flexibus company combines flexibus routes, and an UMS combines a Flexibus company and other means of transportation. The higher the level of abstraction, the wider the scope of adaptation.

UMS Collective Adaptation. In this paragraph, we describe a realistic running scenario, by adding examples about the typical challenges of collective adaptive systems. In turn, it shows how entities can collectively solve problems more efficiently with respect to the case in which each entity adapts by itself.

Let consider a flexibus user that has already caught the flexibus that is, she has officially joint the collective made by the *flexibus driver*, the *flexibus route* and all the *passengers*. Moreover, suppose that two passengers are already on the flexibus, while two other, namely John and Mary, are waiting at different pick-up points on the flexibus route. As we said before, entities,

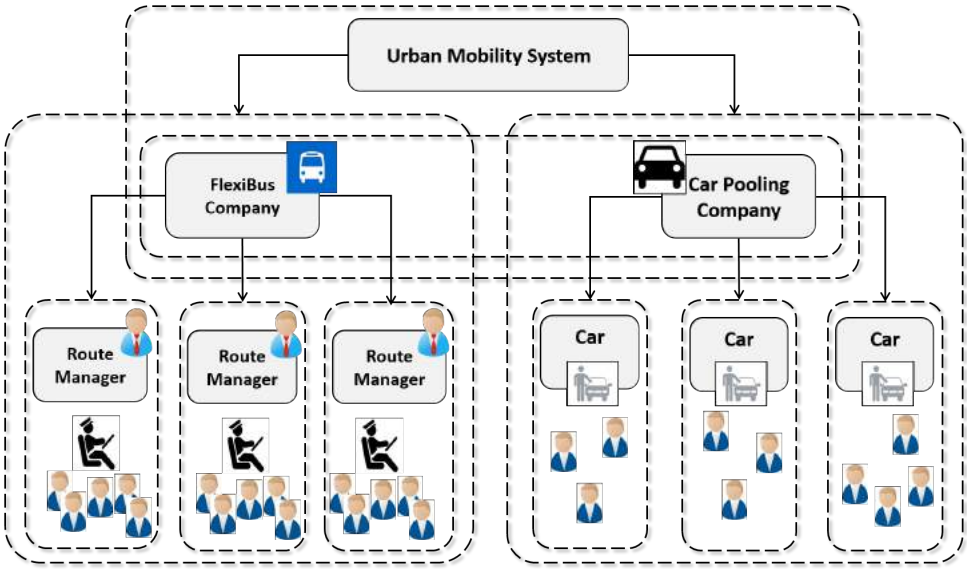


Figure 7.14: Urban Mobility System: examples of ensembles (with dotted lines).

and thus collectives, operate in dynamic environment, whose evolution may affect their behaviors.

Example. *For instance, suppose that, in our scenario, a road on the route of the flexibus gets interrupted, because of an accident. This is the case in which an unpredictable situation affects different entities (e.g., the entities belonging to the collective just mentioned). In this case, the flexibus driver may need to find an alternative route and this can be longer that the current one, thus causing to the passengers on board to be late to their destinations. Moreover, it could result difficult or too expensive in terms of time and money to reach the passengers which are waiting at the pick-up points; at the same time, they also need to find a way to move and terminate their journeys in both cases in which the flexibus can reach them but late or the flexibus cannot reach them at all.*

This situation shows how an unpredictable event, as those occurring daily in big cities, can cause a chain of adaptation needs that must be managed. In the following we give an example of collective adaptation dynamically defined for managing the whole situation by ensuring as little effort and inconvenience as possible for the end users.

First of all, as soon as the *flexibus driver* notices the interrupted road, he notifies the *flexibus manager* about this problem. At this point, two

main situations can occur:

- the *flexibus manager*, by interacting with the *route manager*, can find an alternative route for the flexibus, which comprises the pick-up points where John and Mary are respectively waiting. Moreover, this solution requires not much additional time and we suppose that the passengers on board agree with it;

- the *flexibus manager* is not able to solve the problem, since every possible solution results to be too expensive in terms of time and/or money both for the flexibus company and for the passengers on board, which indeed do not agree with them. In this latter case, the *flexibus manager* can only report the problem to the *UMS*.

We can notice how, in both cases, the final decision is made by a collective interaction among the involved entities (e.g., on board passengers, waiting passengers). In any case, the final adaptation of each entity's behavior is collectively defined. Going back to the collective scenario, suppose that the *UMS* intervention is needed. It is notified about the issue by the *flexibus manager*. Since it is located at the root of the hierarchy, it has a wider view on the whole system, thus it knows about the availability of different transportation means (e.g., car pool), which can be involved in the issue's resolution.

Example. *For instance, suppose that a car pooling having the same destination of the flexibus is already running, and it is made by the driver and a passenger, Lisa. It represents another example of collective. The car pooling route passes not so far from where John and Mary are waiting for the flexibus. A possible solution that the UMS can propose is to switch John and Mary on the car pooling collective, while leaving the flexibus manager to re-plan a flexibus route to the destination, according to only the driver and the on board passengers of the flexibus. However, this solution can cause both to Lisa, to the car pooling driver and to John and Mary a greater investment of time. Thus, in order to make this solution suitable, an agreement has to be find between all the involved users. This is the case in which a collective interaction starts, where the UMS acts as*

an intermediary entity asking for the acceptance of the alternative solution to the users. Supposing that the alternative solution fits with all the users requirements (e.g., in terms of arrival time) and a final agreement is found, the collective adaptation is performed. Eventually, John and Mary will share the car pooling with Lisa, while the flexibus passengers will reach the destination by following a different route. Moreover, a reimbursement must be managed for John and Mary, if they already paid for the flexibus.

In conclusion, this scenario represents just one of many situation in which an unpredictable event affects the execution of collective processes, thus triggering collective adaptations.

7.3.2 Adaptive by Design Urban Mobility System

In this section, we give a (partial) overview of the design of the UMS just described. The system is modeled through a set of *domain objects* representing the above-mentioned system's entities and implementing different behavior, described by *domain properties*.

In Figure 7.16 we report a portion of the UMS model, with the *soft dependencies* among domain objects. The `Urban Mobility System` represent the interface with the users. It can partially define the functionality for the multi-modal planning of trips. Different entities can join the system and publish different planning procedures. At run-time, when the user needs and preferences are known (e.g. preferred transportation mean), the `Urban Mobility System` will use the fragments offered by the suitable planners to specialize its abstract activity and to eventually provide possible alternatives to the user. In this way, the `Urban Mobility System` will dynamically interact with the other domain objects in Figure 7.16 to manage the organization and execution of the user trip, based on her selected travel alternative.

As stated in chapter 4, the dynamic features offered by the framework rely on a set of concepts, describing the operational environment, on which each domain object has a partial view (e.g. flexibus trip, route passenger status, handle route). We give some examples in Figure 7.15. Consider for instance the domain property `Route passenger status` in Figure 7.15 that models the typical daily trip of a citizen using the flexibus mobility service

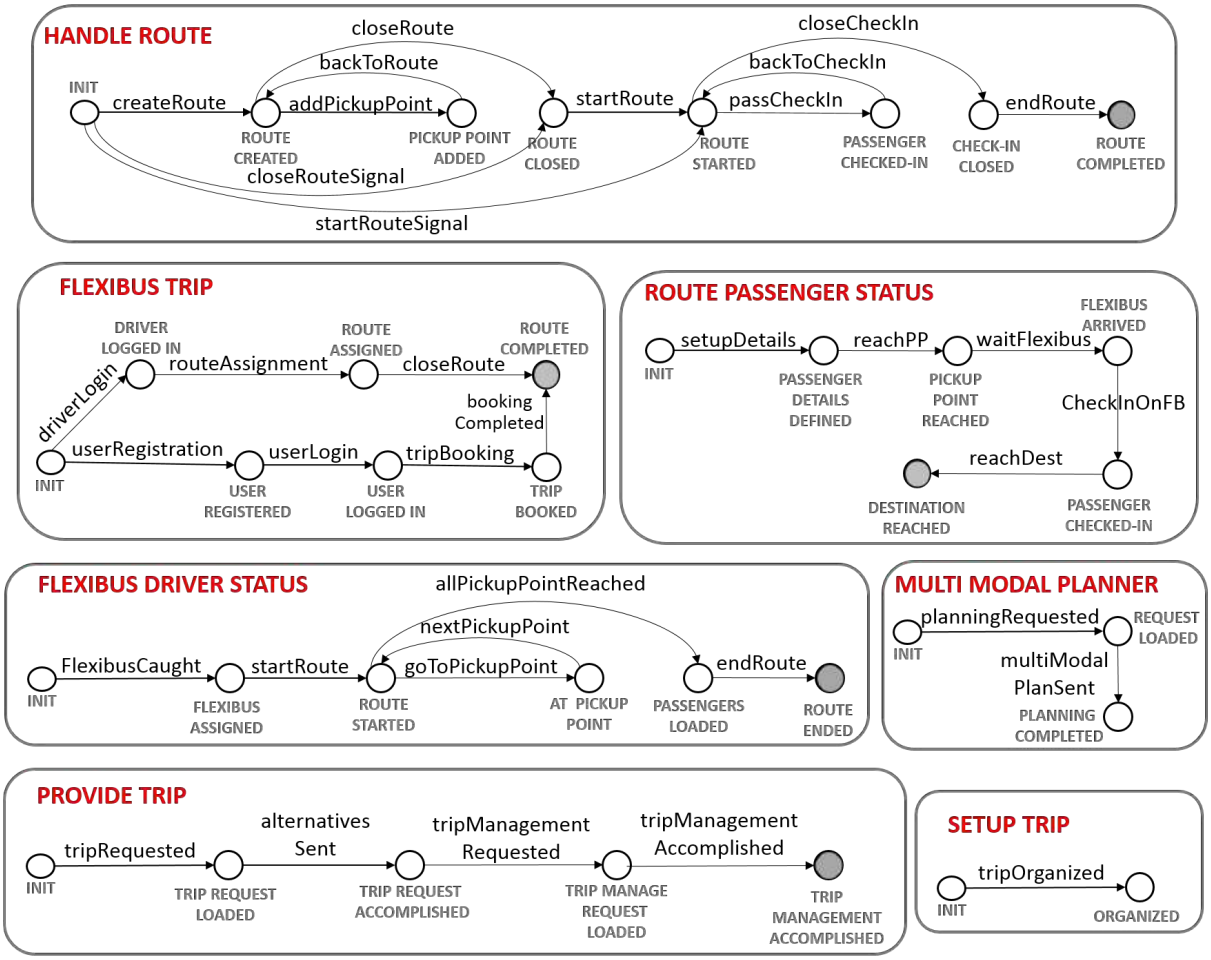


Figure 7.15: Domain properties modeled as STS.

offered by the UMS. The journey of the user needs to be defined in all its details (PASSENGER DETAILS DEFINED), than the user reaches the pick up point (PICK-UP POINT REACHED) and she waits for the arrival of the flexibus (FLEXIBUS ARRIVED). During the journey she is in the PASSENGER CHECKED-IN status (this is used to model the fact that the user caught the flexibus) and she eventually reaches the destination (DESTINATION REACHED).

We recall here that activities in domain objects processes and fragments will be annotated with *goals*, *preconditions* and *effects* defined on top of domain properties. These annotations will drive the dynamic services selection and composition, enabling a run-time chain of refinements, as discussed in chapter 5.

The resulting adaptive system is a dynamic network of domain objects.

We can see in Figure 7.16 how soft dependencies are established between a domain object and all those domain objects in the system whose provided functionality (internal domain knowledge) matches with one of its required behaviors (domain property in its external domain knowledge). During the system execution, a soft dependency between two domain objects becomes a *strong dependency* if they inter-operate by injecting and executing a fragment.

Furthermore, in our approach based on the extended domain objects model presented in chapter 6, a *role* is modeled as a domain object, with its internal behavior and its exposed fragments modeling the operation of participating entities in the scope of the ensemble to which the role belongs to. For instance, in Figure 7.16, the domain objects modeling roles are depicted in grey (e.g., Flexibus Company, Route Passenger, Flexibus Driver, Route Manager). Besides, each role specifies, among its fragments, the fragment (in blue in Figure 7.16) that allows a domain object to adopt that role, through a specialization process. More specifically, when a domain object refines an abstract activity while executing its process by using this kind of fragment, this specialization signs the entrance of the domain object into an ensemble by playing that role (e.g., the create route fragment in the Route Manager domain object allows the Urban Mobility System domain object to play the role of Route Manager in a Route Ensemble).

Finally, to enable collective adaptation, each domain object implements a set of collective adaptation *solvers*, as well as a set of collective adaptation *handlers*, as specified in chapter 6. We recall that solvers model the ability of a domain object to handle one or more *issues*. Since the environment changes frequently and unpredictably, the system requires constant monitoring. Handlers are used to capture issues, during the nominal execution of a domain object, and to trigger the appropriate solver. Moreover, each handler refers to a finite scope in the process of a domain object. We refer to chapter 6 for examples of solvers and handlers in the context of the UMS.

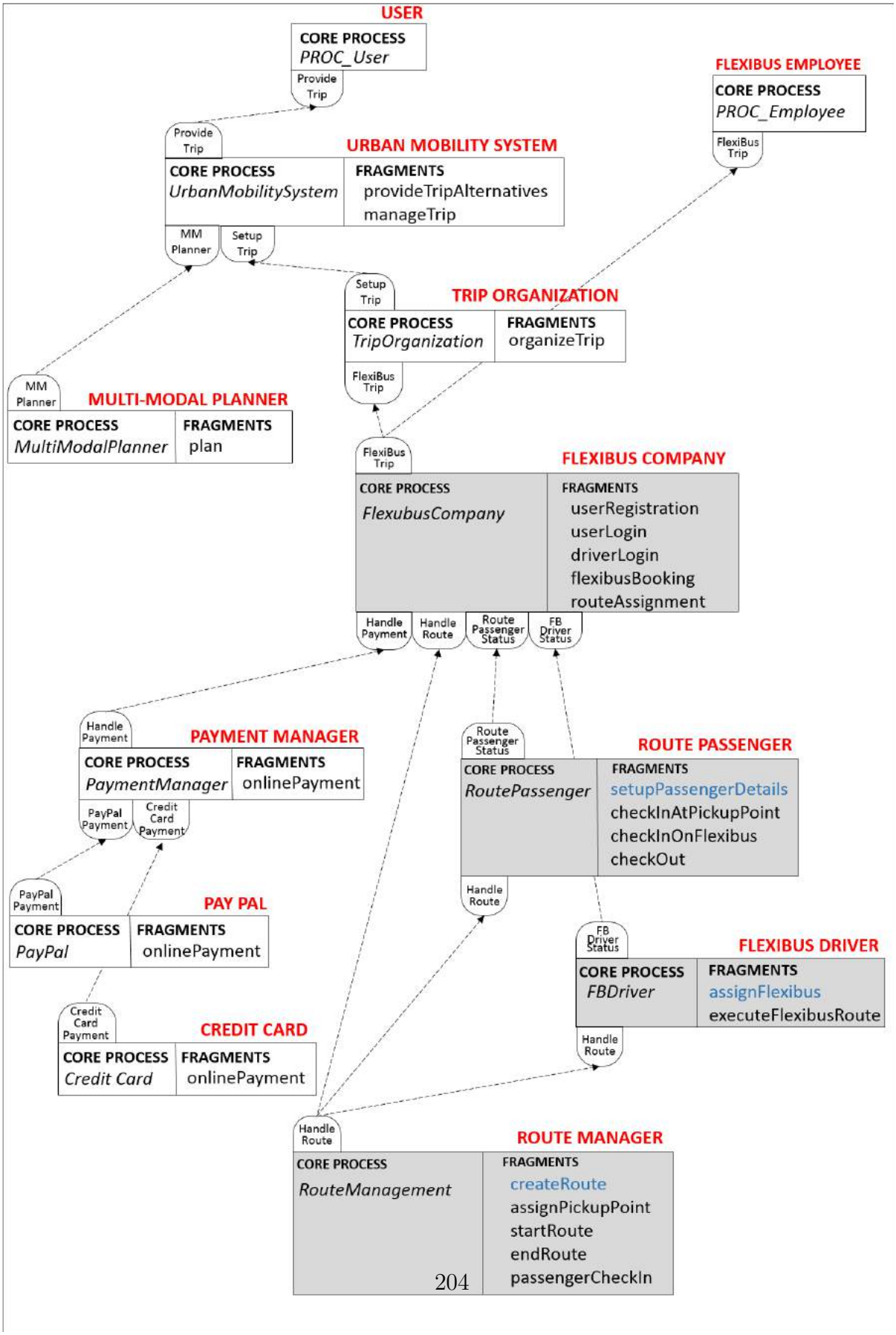


Figure 7.16: Portion of UMS with soft dependencies.

7.3.3 DeMOCAS Implementation

The design for adaptation framework has been implemented as depicted in Figure 7.1. In particular, the Figure distinguishes, by green-coloring them, the components referring to the modeling and execution of CAS. These components are the *Collective Adaptation Manager*, the *Ensembles Manager*, the *Ensembles Models* and the *Ensembles Repository*, implementing the collective adaptation approach and whose functions have been widely discussed both in chapter 6 and in this chapter.

These components are all exploited by DeMOCAS for the design, execution and collective adaptation of the UMS. Indeed, by running DeMOCAS it is possible to (i) follow the normal behavior of the UMS; (ii) inspect in real time the process execution of each involved domain object, together with the evolution of its domain properties; (iii) inspect the dynamic ensembles formation and their life-cycle and, finally, (iv) trigger issues in order to inspect the application of collective adaptations solutions.

The following paragraph is devoted to describe the main interface of DeMOCAS, how it is organized and how it is connected with the beforementioned functionalities.

DeMOCAS Interface. In this section we describe, from an high level point of view, the functioning of DeMOCAS, by highlighting and describing its main functionalities. In general terms, DeMOCAS starts by loading and reading a scenario, such as the UMS, defined in a `*.xml` file and it interprets it, by reading all the scenario related artefacts (e.g., domain objects, fragments, processes) types and instances with the corresponding properties (e.g., preferences in the case of users domain objects) based on their types. Figure 7.17 shows the main window of the demonstrator. In particular, the upper side of the window displays a list of domain objects instances [18] (e.g., `User_1` represents the currently running user) and their position on the map. The lower side, instead, displays both the process model and the process execution for the current running instance, with all the corresponding information, such as, domain object instances of the

[18] In the ALLOW Ensembles EU project, in whose context the DeMOCAS Demonstrator has been developed, the domain objects have been used to model the so-called *cells*. For this reason on the screenshots of the demonstrator interface you can find the Cell term. In this dissertation we use the term domain objects, without loosing of generality, since cells are modeled as domain objects.

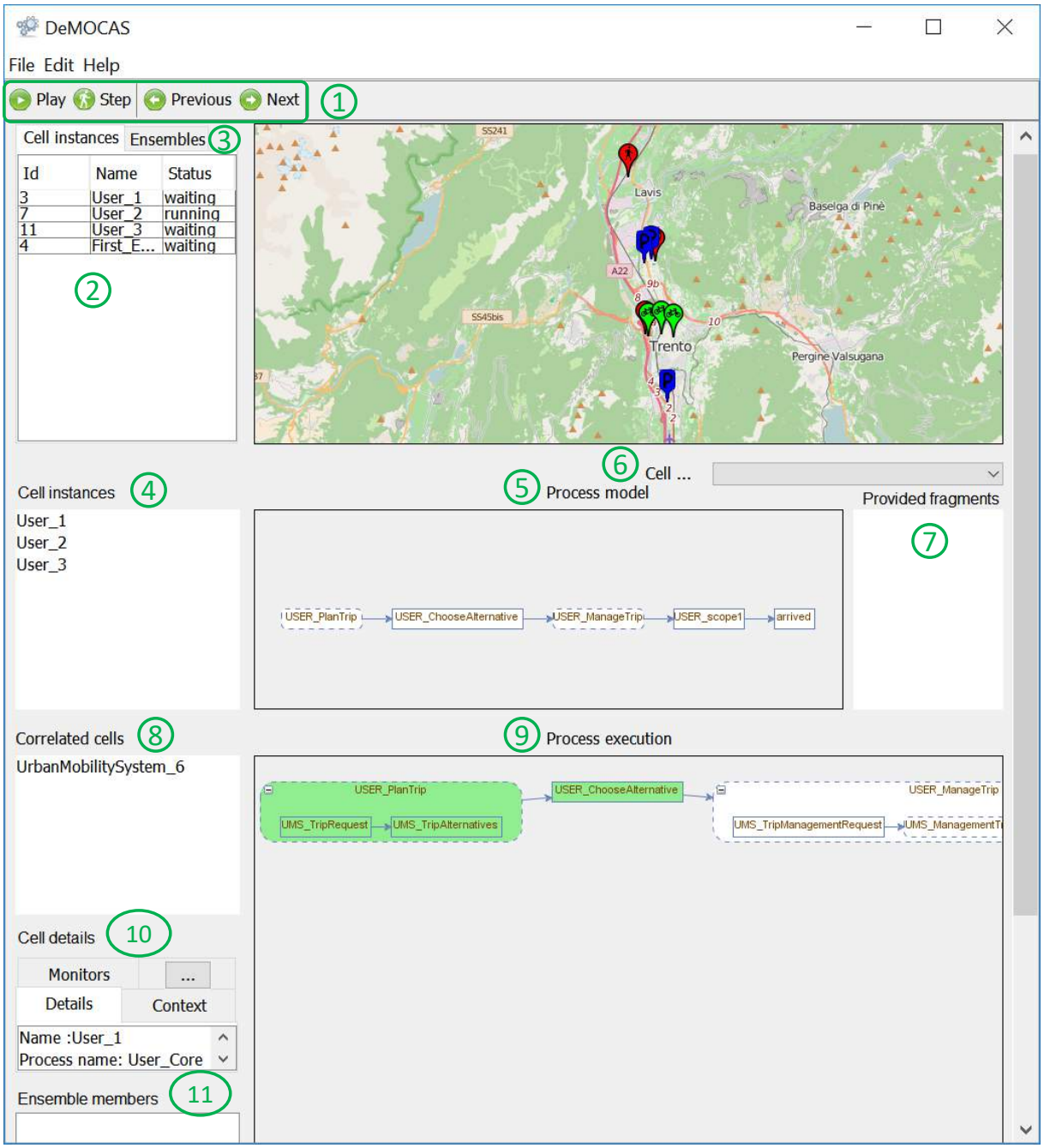


Figure 7.17: DeMOCAS GUI - Main Window.

same type, correlated domain objects and domain object details.

The main window of DeMOCAS allows users to interact with the demonstrator and follow the run-time execution of the loaded scenario. In detail, it is organized as listed in the following. The numbers in the list correspond to the numbers labeling the screenshot in Figure 7.17.

1. **Toolbar**: it contains the buttons for the management of the scenario execution, namely:

   - **Play**: it allows users to start and execute the current scenario. In this case the demonstrator calls the `Process Engine` which is in charge of executing the domain objects processes.
   - **Step:** it is used to force the `Process Engine` to execute the scenario's processes step by step. This allows DeMOCAS users to analyze and inspect the run-time execution.
   - **Previous and Next:** they allow users to navigate through domain objects instances and update the interface accordingly.

2. **Cell Instances:** it lists the domain objects instances. If an instance is selected, the interface is updated accordingly (e.g., by loading the corresponding process model, the process instance, its provided fragments and its correlated domain objects). The Map in the center of the interface will highlight the current position of the selected domain object instance.

3. **Ensembles:** this tab allows users to select among the different available ensembles from the list, by also highlighting their members on the map.

4. **Cell instances:** this area displays all the domain objects instances of the domain object model currently selected (see point 2).

5. **Process Model:** this area shows the process model of the current domain object instance. In the figure we can see the abstract activity `UMS_CalculateTripAlternatives` that will be refined during the run-time execution of the process.

6. **Cell models:** it allows the selection of a domain object model from a drop down list and the visualization of its corresponding process model.

7. **Provided fragments:** here the provided fragments of the current domain object are shown.

8. **Correlated cells:** when a domain object communicates and exchanges knowledge with other domain objects a correlation among them is created. All these correlations are listed in this area of the interface.

9. **Process execution:** The interface shows the process instance in execution, by green-coloring the executed activities, for each process instance. If the process instance contains abstract activities (represented with dotted lines), they are refined at run-time by using fragments provided by other domain objects. The refinement is graphically shown by the injection of the fragments composition into the abstract activity, as in the case of the `UMS_CalculateTripAlternatives` activity in Figure 7.17.

10. **Cell details:** it is a view tab containing information about the current running domain objects. In particular, the following tabs are showed:

    - **Details:** it contains the general details about a domain object instance (e.g., instance name, process name).
    - **Context:** it contains the current knowledge values for the domain object instance, which evolves during the execution time.
    - **Monitor:** it contains the monitors (i.e., *process scopes* with relative *handlers*) defined in a domain object process, if any.
    - **More details button:** for special domain objects, like those modeling users, there are a lot of information that the demonstrator's users can verify and change.

11. **Ensemble members:** if the current domain object belongs to an ensemble, the other ensemble members are listed in this area.

The bottom part of the main window relates to the execution of domain objects processes, through their dynamic adaptation, specifically performed via the application of the refinement mechanism. For the CAS execution, DeMOCAS behaves exactly as stated for ATLAS, in its Runtime Execution tab, described in section 7.2.3. In the following paragraph, instead, we focus on the Collective Adaptation Manager interface.

Collective Adaptation. As described in chapter 6, during the execution of an ensemble, each role can produce issues and can take care of issues produced by others (i.e., *issue solver*). Issues generally correspond to different extraordinary situations that can happen to a role and that can be reported to others. For each *issue instance* an *issue communication* is created and it is used to send an issue to partners that are supposed to resolve it. While the issue communication is a way to propagate resolution activities between partners, the *issue resolution* corresponds to the high-level model of internal elaboration being done by role instances. When an issue is raised, each role instance in an ensemble may resolve it locally or send one or a few issues to the other partners as a part of resolution procedure. This procedure is represented as a tree (i.e., *issue resolution tree*) where nodes are issue resolutions or issue communications, the root node is an issue resolution, leaves are issue resolutions, all children of an issue resolution are issue communications, and all children of an issue communication are issue resolutions.

One of the added values of DeMOCAS, with respect to ATLAS, is the implementation of the *Collective Adaptation Manager* component, in the component diagram of Figure 7.1. It deals with the resolution of dynamically arising issues, by triggering the collective adaptation algorithm and managing the communications among the involved domain objects. The graphical interface of the Collective Adaptation Manager is reported in Figure 7.18. It is composed of two parts illustrating the issue resolution procedure in an ensemble. Figure 7.18 depicts a screenshot of the overall window.

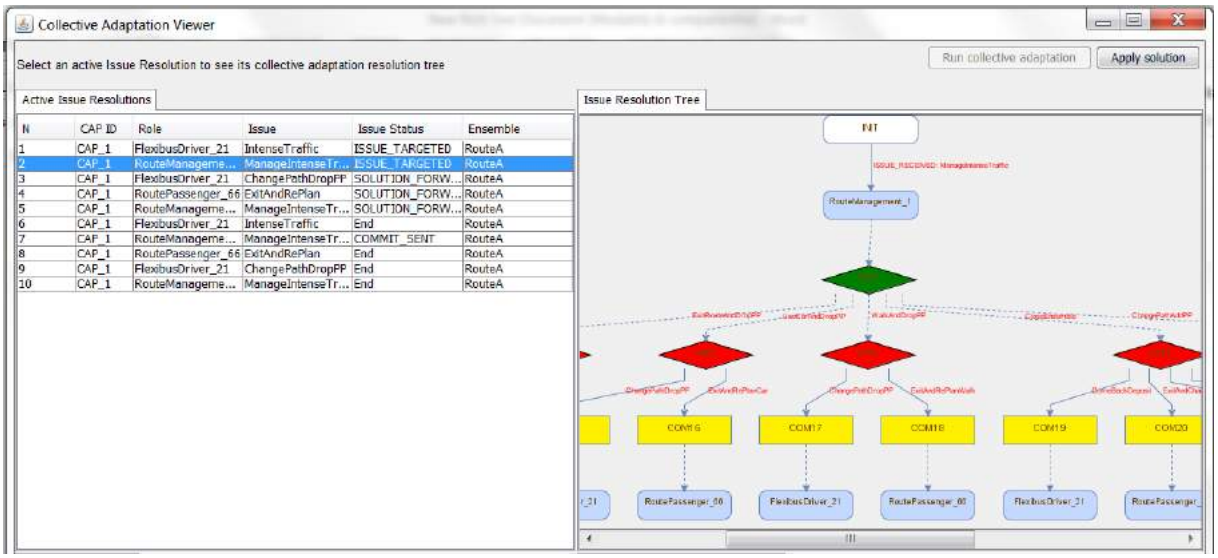


Figure 7.18: Graphical Interface of the Collective Adaptation Manager.

The left side shows the list of issue resolution managed by the Collective Adaptation Manager with their information on the *problem ID*, the *Role* involved, the *Issue Type*, the *Status* of the current issue and in which *Ensemble* it is resolved.

The right side, instead, shows the issue resolution tree derived from the collective adaptation algorithm. Indeed, the *issue resolution process* generates a resolution tree, modeling all possible solutions for a given issue, over which the *best solution* is selected. The tree depicted in Figure 7.18 shows the role from which the issue has been triggered and all the sub-trees generated during the resolution. Each sub-tree represents a possible solution (i.e., solver) for the initial issue triggered. When an AND node is generated, it means that a specific solution has triggered extra issues to be solved. Instead, when an OR node is generated, it means alternative solutions are applicable for the same issue. This is the case in which we exploit the multi-criteria ranking approach, based on analytic hierarchy process (AHP) [158], to select the best alternative.

7.3.4 DeMOCAS Evaluation

DeMOCAS exploits the modeling approach described in this dissertation, and further extended to deal with CAS, as described in chapter 6. *Adapt-*

ability and *context-awareness* are key embedded characteristics of the targeted systems. Moreover, with DeMOCAS we can (i) simulate the UMS behavior (citizens, drivers, managers of mobility resources), (ii) run the execution and adaptation of service-based processes attached to domain objects in the UMS system and (iii) inspect in real time the execution state of the whole system (domain object instances, current state of their processes and domain properties).

The specification of the UMS we used to evaluate our approach, whose overview is reported in section 7.3.1, contains 19 domain object models, 37 fragment models and 17 types of domain properties. We evaluated our techniques using a dual-core CPU running at 2.7GHz, with 8Gb memory. We carried out an experiment in which we run the demonstrator, simulating the operation of the UMS, and we collected the execution and adaptation statistics such as the number of adaptation cases, the complexity of each adaptation problem and the time required to generate an adaptation solution. We considered 10 runs of the system, each having 32 *User* instances with *different* preferences, transport needs and itineraries. For each run, more than 400 refinement cases were generated. Figure 7.19 shows the distribution of problem complexity considering the 10 runs.

The complexity of a problem is calculated as the total *amount of transitions* in domain properties and fragments present in the problem. For simplicity in the graph we aggregated the problem complexities in ranges of 5. That is, the bar with complexity 5 contains all the problems whose complexity is included in the range $[1, 5]$, and so on. For most problems (89%) the complexity is below 50 transitions, with 2 to 4 fragments and 1 to 4 domain properties in the problem scope. The most complex problems (around 3%) have a complexity of 75 transitions, with 7 fragments and 7 domain properties.

Figure 7.20 shows the percentage of refinement problems solved within a certain time. We can see that, for all the runs, 89% of problems are solved within 0.2 seconds. Only 3% of the problems require more than 0.5 seconds to be solved, and the worst case is anyhow below 0.9 seconds.

Finally, Figure 7.21 relates the (average) time required to solve a composition problem to the problem complexity. The average time is computed considering in the 10 runs all the refinement problems having the same com-

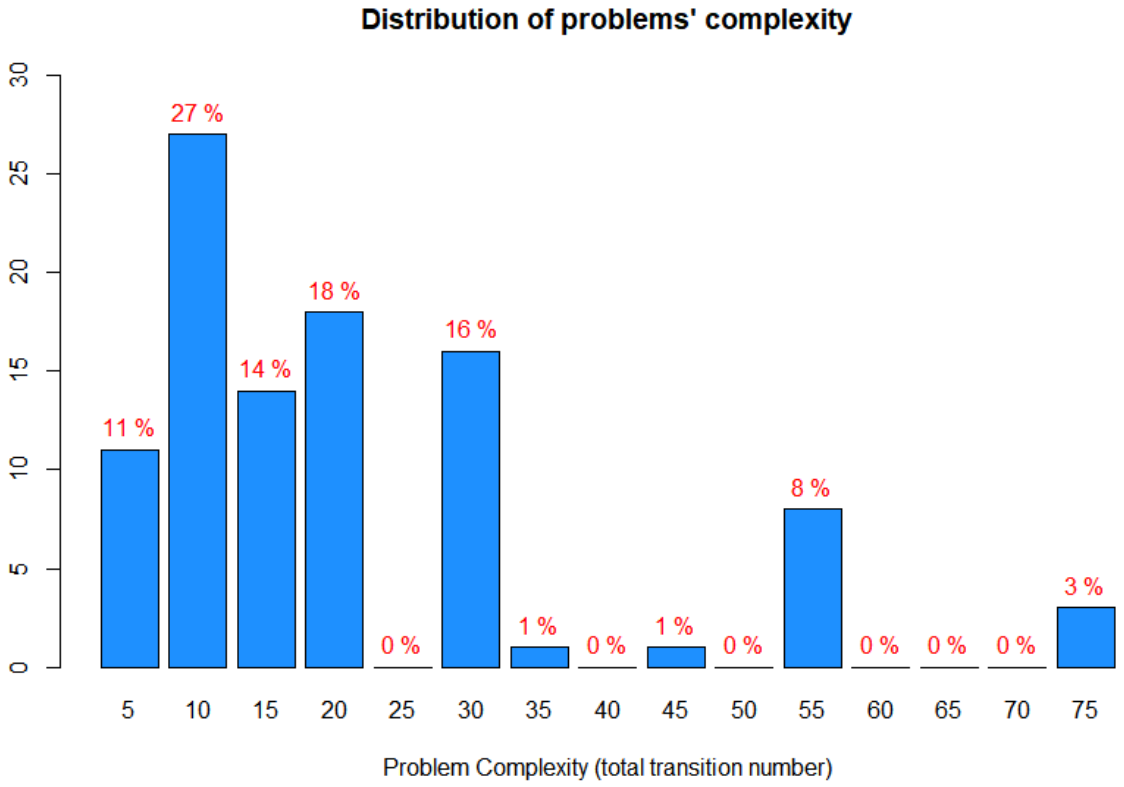


Figure 7.19: Distribution of problems' complexity.

plexity in terms of total transitions. As expected, problems with higher number of transitions (and hence the most complex planning domain) take more planning time than problems with less complexity.

These results demonstrate the *usability* and *scalability* of our approach when applied to a real-world complex scenario.

7.4 CAStlE: a Domain Specific Language for Engineering Collective Adaptive Systems

In chapter 6, we have proposed a framework to build CAS that integrally addresses the problem of collective behavior and hierarchical organization of ensembles, allowing for flexible and efficient adaptation in such systems. The framework relies on the design for adaptation approach presented in this dissertation, mainly based on the domain objects model and the incre-

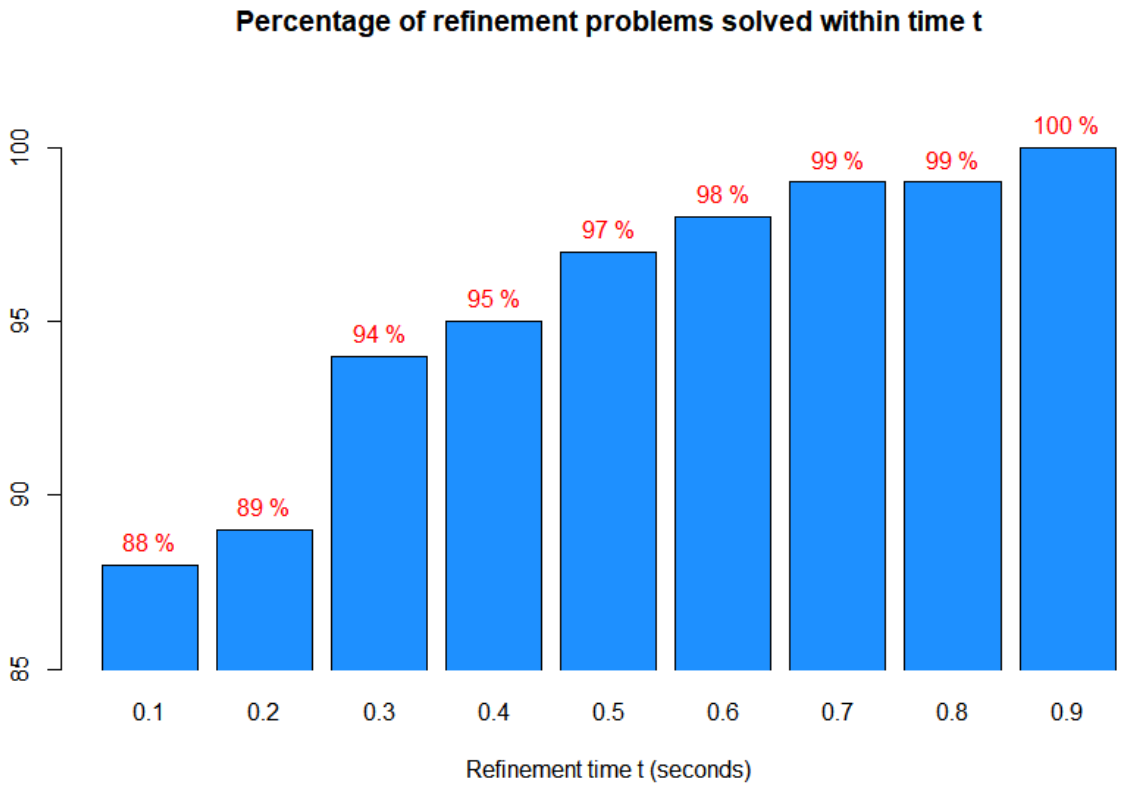


Figure 7.20: Percentage of problems solved within time t.

mental service composition approach, for modeling and execution service-based systems, respectively. Moreover, it has been built around the concept of ensembles role, primarily determined by the ways a role negotiates with other roles. Although we provide approaches for the design and operation of (collective) adaptive systems, as well as techniques for performing both *selfish* and *collective* adaptations, a sound engineering process for CAS is still missing. In particular, a CAS is specified at a low level of abstraction (XML files), a task that tends to be time-consuming and error-prone when the size of the system grows. Moreover, the comprehensibility of a systems specification is hindered.

We adopt the Model-Driven Engineering (MDE) [164] vision, which proposes to reduce the complexity of development by adopting models as first class artifacts in the process. This would allow developers to tackle the engineering process at a higher level of abstraction. The modeling sup-

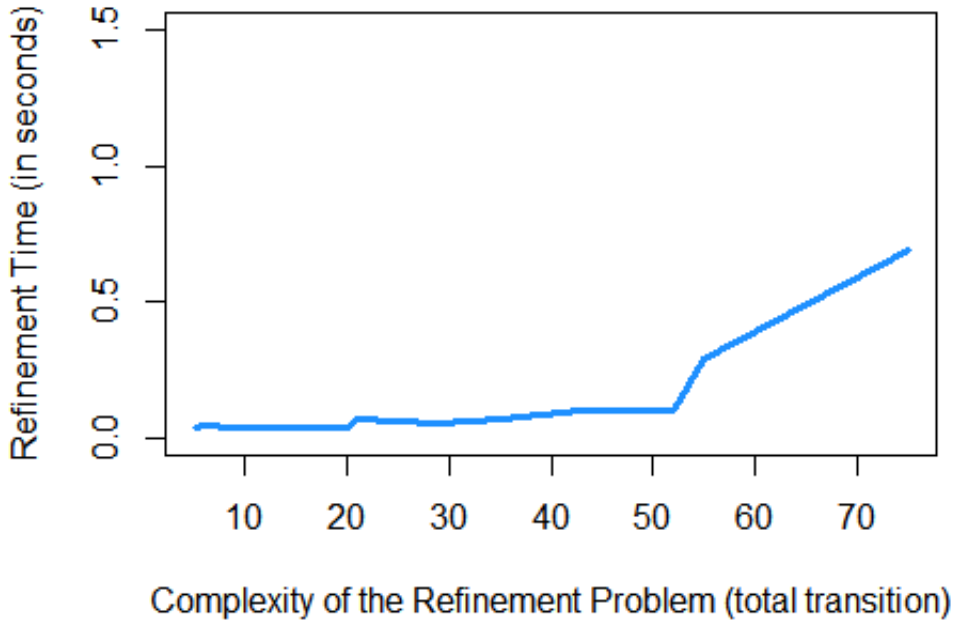


Figure 7.21: Average refinement time for problem complexity.

port should be tailored to a CAS as a whole, covering all its aspects, from single-entity to ensembles modeling, and from selfish to collective adaptation constructs modeling. Therefore, we extended our framework by specifying a domain specific language (DSL) for defining CAS, named CAStlE [32, 33]. The DSL is made-up of three different views, devoted to the specification of an *adaptive system*, *ensembles*, and *collective adaptation*, each of which based on a corresponding meta-model. This modeling support allows the reduction of the complexity of specifying CAS by providing domain-specific concepts to designers, who do not have to deal with XML files. Moreover, it avoids inconsistencies and other specification errors by-construction, since the corresponding models would be invalid against the meta-model definitions. As a consequence, by adding this modeling support layer to our existing design for adaptation framework, we can enhance the CAS design phase by reducing the possibility of incurring in errors, as well as by improving the understandability of system specifications. The

design and implementation of CAStlE is detailed in the next section.

7.4.1 CAStlE Design and Implementation

The design of CAStlE has been tackled by adopting the separation-of-concerns principle [165]; more precisely, three different sub-languages have been introduced, each of which specifically tailored to a specific aspect of the system. In particular, one is devoted to the specification of an adaptive system and its domain objects; one enables the definition of ensembles together with issues and solvers; and one addresses the collective adaptation design, that is event handlers. In this way, the complexity of designing a CAS is alleviated by partitioning the system in simpler sub-problems; moreover, it discloses the opportunity of re-using parts of the system specification in other contexts, notably the same adaptive system in different ensembles, or the same ensemble in different collective adaptations. In the following three paragraph, each of the languages will be presented by illustrating the main modeling concepts offered to the designer[19]. Each diagram contains named boxes that represent meta-classes, possibly containing meta-attributes. Moreover, meta-classes are connected by means of relationships. In particular, a diamond-shaped arrow-head is used for compositions, a triangle-shaped arrow-head for generalizations, and simple arrow heads for associations.

The Adaptive System modeling language. An excerpt of the Adaptive System metamodel is shown in Figure 7.22. An `AdaptiveSystem` is a composition of `DomainObject`s, each of which including a `CoreProcess`, `Fragment`s, and `DomainProperty`s. It is worth noting that the multiplicity boundaries put constraints on the well-formedness of an Adaptive System model. Notably, there must be at least a `DomainObject`, and each `DomainObject` must contain one unique `CoreProcess`. The relationships between domain objects and domain properties establish that a domain property represents `internaldomainknowledge` if defined within the `DomainObject` (composition relation), whereas it represents `external domainknowledge` if referred to by a simple association.

[19] The reader is referred to `https://github.com/das-fbk/CAS-DSL` for the complete metamodels.

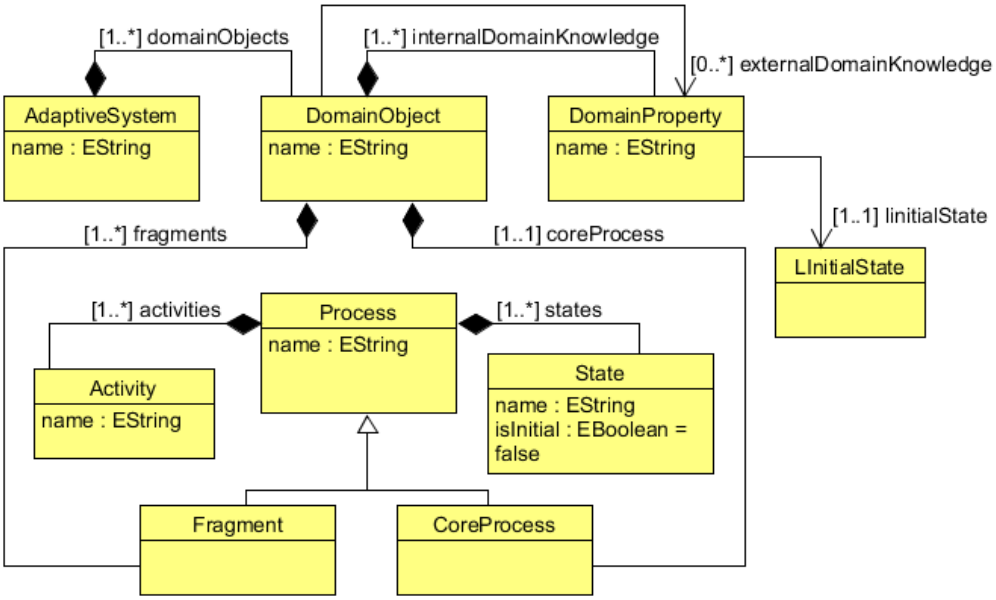


Figure 7.22: Excerpt of the Adaptive System metamodel.

Both processes and domain properties can be reduced to state transition systems. From a modelling point-of-view, the only difference between the two is that for processes (both core and fragments) there is no notion of initial state, or better, it is possible to set multiple states as initial through a Boolean attribute (see `isInitial` in `State`). On the contrary, a `DomainProperty` must have a `LInitialState`, as constrained by the multiplicity boundaries of the `linitialstate` relationship. The remaining part of the modeling concepts mirrors the adaptive system model.

Figure 7.23 shows an excerpt of the Urban Mobility System model, detailed in section 7.3.1 and specified by using CAStlE. As we can see, it is composed by a set of Domain Objects (e.g., User, UrbanMobilitySystem, FlexiBusDriver, FlexiBusCompany, etc.), each one including a `CoreProcess` (e.g., PROC_User, PROC_UrbanMobilitySystem, etc.) and a set of `Fragments` (e.g., UMS_provideTripAlternatives, FD_executeFlexibus Route, FC_userRegistration). The `DomainProperty` modeling the internal knowledge of a domain object is also part of the domain object itself.

Figure 7.24 gives more details on how CAStlE can be used to model `CoreProcesses` and `Fragments`. In particular, the `CoreProcess` of the

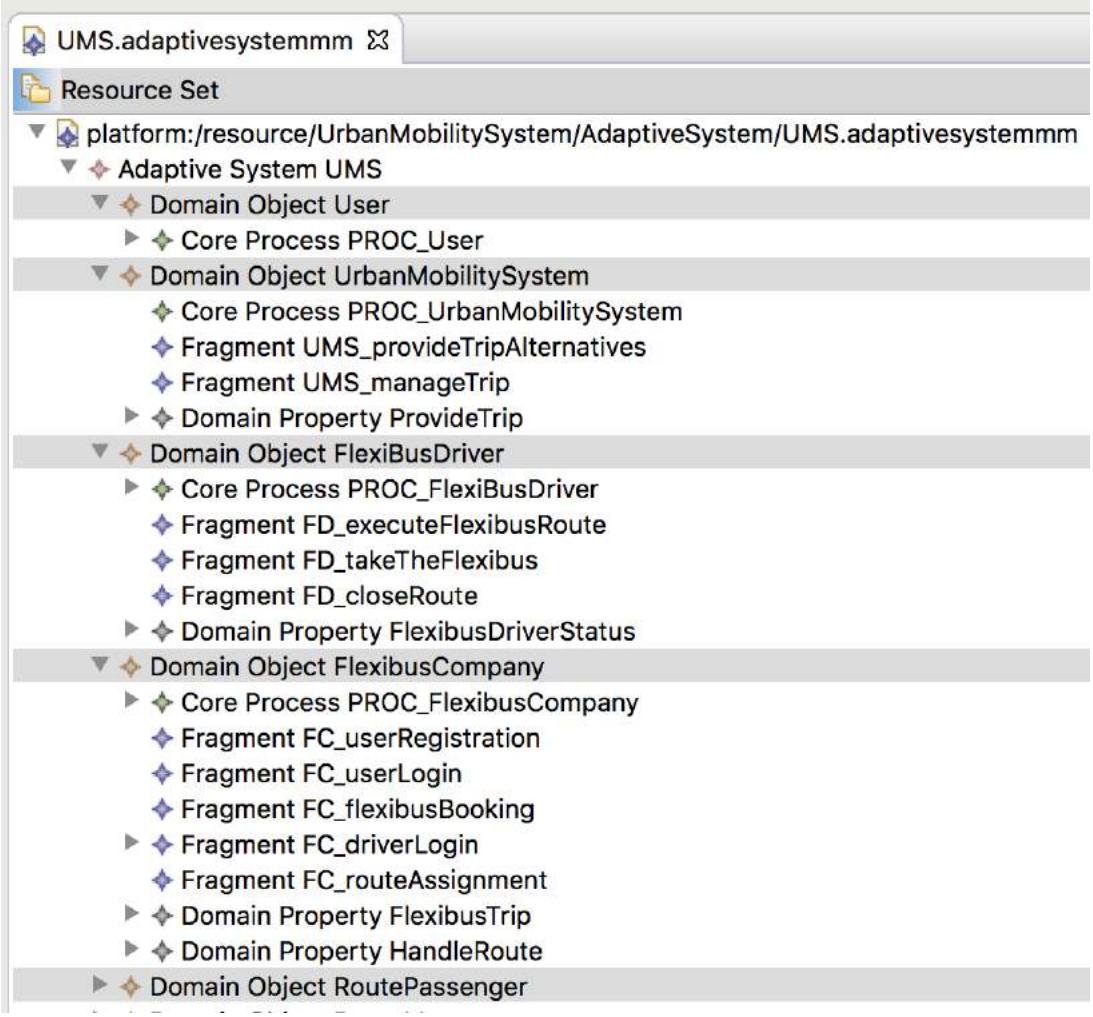


Figure 7.23: Excerpt of the Urban Mobility System.

User is composed by a sequence of activities (i.e., abstract and concrete) and a set of goals that are used to annotate the abstract activities (e.g., USER_PlanTrip, USER_ManageTrip and USER_ExecuteTrip). A `Fragment` is modeled as a state transition system, with states (e.g., ST0, ST1, etc.) and the respective input and output activities (e.g., DriverLoginRequest, DriverLoginAck in the fragment FC_driverLogin).

The Ensemble modeling language. As previously discussed, domain objects can be grouped in ensembles in order to play a specific role. This is reflected in the ensemble modelling language, an excerpt of which is shown in Figure 7.25. In particular, an `Ensemble` contains a number of `Roles` played by `DomainObjects` and defines a set of `Issuetypes`. The

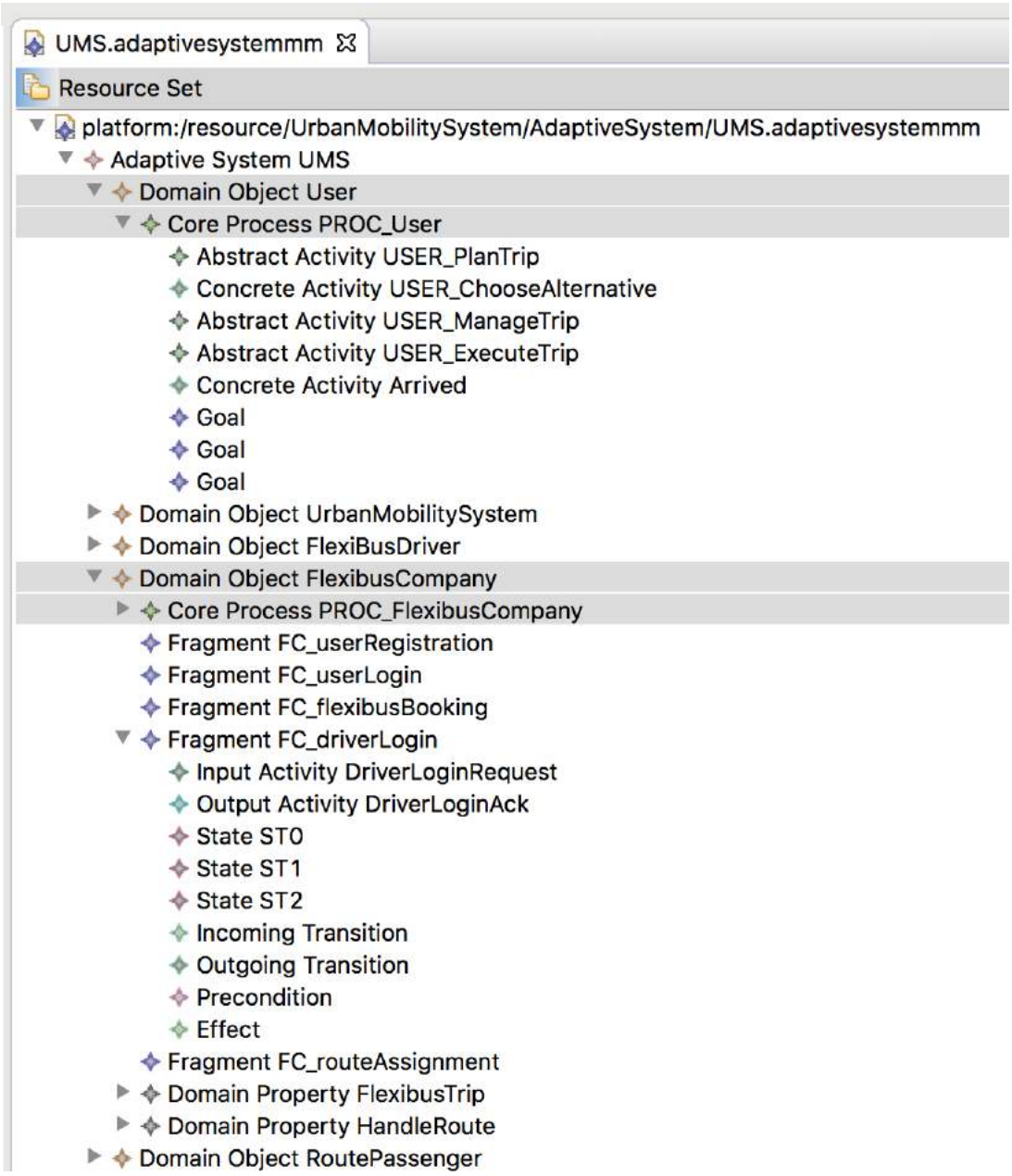


Figure 7.24: Excerpt of Core Processes and Fragments.

relationships between domain objects and roles allow for a domain object to play multiple roles (or none), while a role can be played by exactly one domain object.

A role is characterized by `RoleParameter`s and `Preference`s that describe the state of the role and domain-specific preferences, respectively. A role can define `Solver`s and trigger `IssueType`s, as represented by the

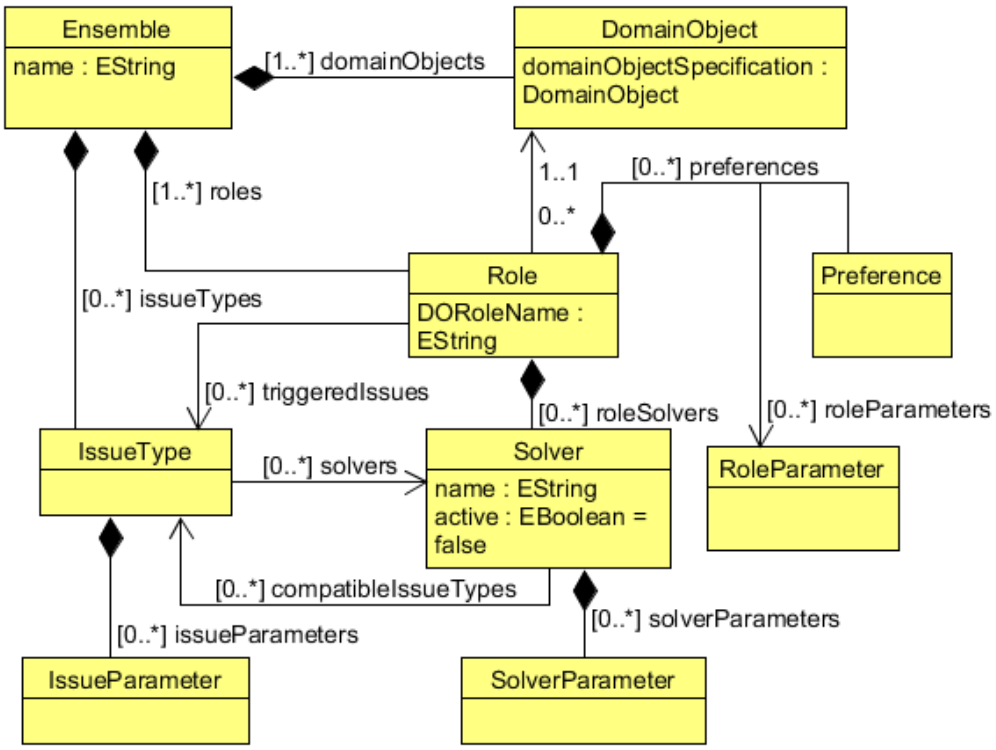


Figure 7.25: Excerpt of the Ensemble metamodel.

`roleSolvers` and `triggeredIssues` relationships, respectively. Both issues and solvers can have parameters, and are related to each other. In particular, a solver is associated with compatible issue types and issue types are associated with corresponding solvers.

At this point it is important to remark that in general `DomainObject` elements raise a consistency issue, since they shall be defined in the adaptive system model to be used in an ensemble. This problem is smoothly solved by the DSL through the link defined in the `DomainObject` entity in this meta-model, called `domainObjectSpecification`, which points to a `DomainObject` entity in the adaptive system language as presented in the previous section. Such a link guarantees *by-construction* that each domain object used in an ensemble model has been defined in the adaptive system model.

In Figure 7.26 we report the FlexiBus Route Ensemble model, to give an example of how an ensemble can be modeled by using CAStlE. It is

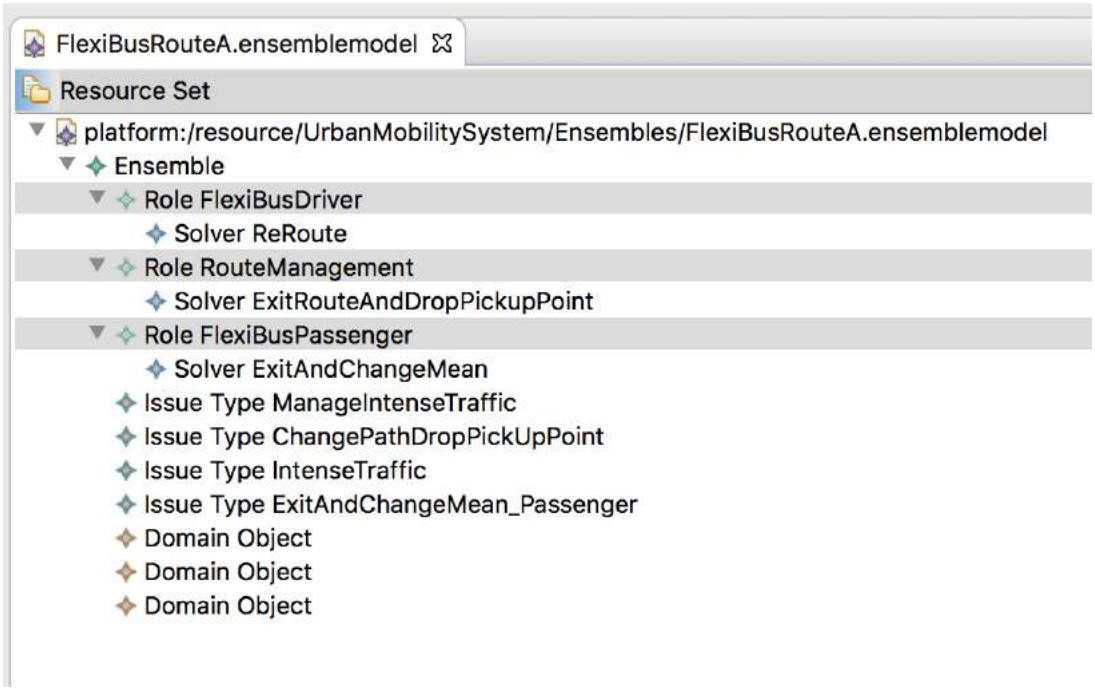


Figure 7.26: Excerpt of FlexiBus Route Ensemble.

composed by a set of `Role`s (e.g., FlexiBusDriver, RouteManagement and FlexiBus Passenger), each one providing a specific `Solver` (e.g., ReRoute, ExitRouteAndDropPickupPoint, and ExitAndChangeMean) able to solve or to generate an `IssueType` (e.g., ManageIntenseTraffic).

The Collective Adaptation modeling language. Starting from domain objects grouped into ensembles, it is necessary to define adaptation strategies to realize a CAS. Therefore, the collective adaptation language (depicted in Figure 7.27) establishes a different grouping layer over domain objects, tailored indeed to collective adaptations. In this respect, a `CollectiveAdaptation` contains a number of `EventHandler`s defined over a set of `RoleActivity`s, i.e. the activities which are in the *scope* of the handler. Each handler catches `Event`s, that are instances of issue types, and triggers `Adaptation`s, that are instances of solvers.

Also in this case it is worth to note that `IssueType` and `Solver` entities in this model must be consistent with what is defined in the ensemble model. Moreover, `RoleActivity`s have to be consistent with domain object activities defined in the adaptive system model. Similarly to the ensemble meta-model, the links `issueSpecification`, `solverSpecification`,

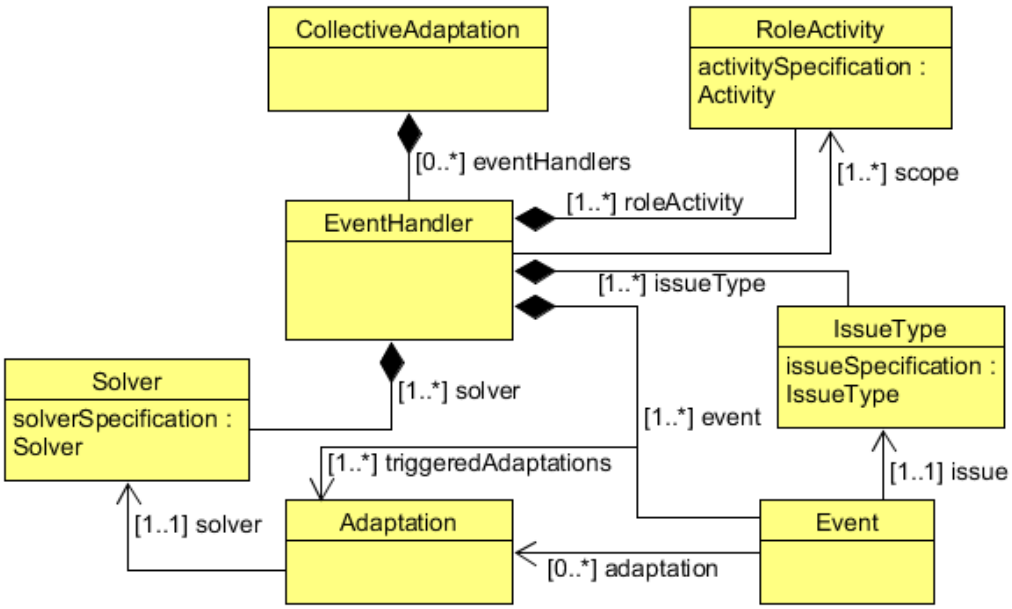


Figure 7.27: Excerpt of the Collective Adaptation metamodel.

and `activitySpecification` preserve by-construction the consistency between the three different views of a collective system specification.

In Figure 7.28 we show the model of an `EventHandler`, namely `FlexiBus Delay`, as defined in CAStlE. Its scope is made by an abstract activity (e.g.,

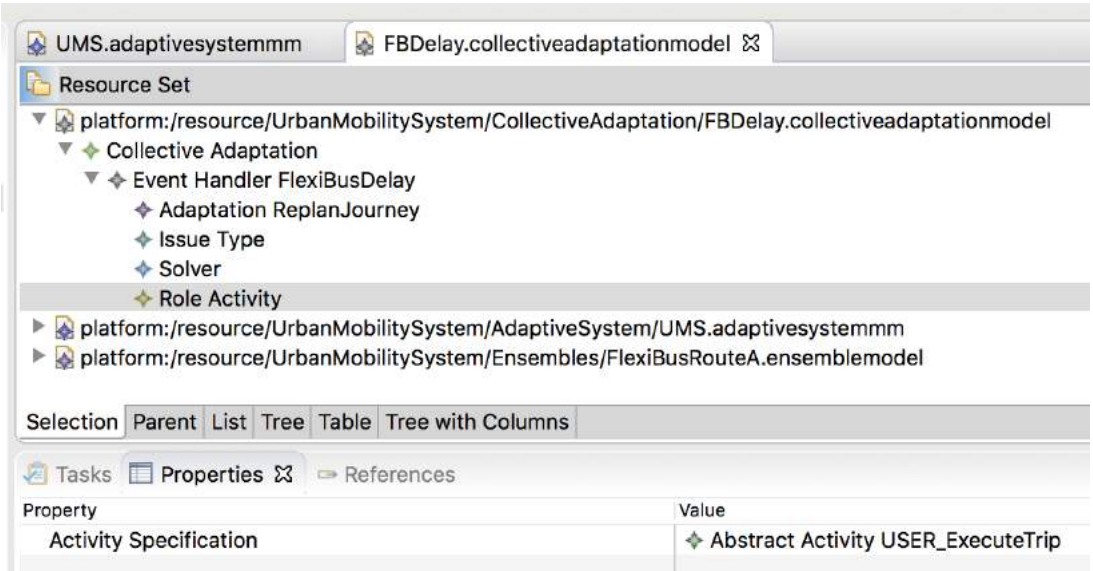


Figure 7.28: Excerpt of FlexiBus Delay Collective Adaptation.

USER_ExecuteTrip) in the User core process. It specifies the adaptation that will be triggered (e.g., ReplanJourney) when the corresponding issue is caught.

Tool Support. As a proof-of-concept for the language we implemented CAStlE through the Eclipse Modeling Framework (EMF)[20]. EMF supports MDE techniques with a set of tools for creating modeling languages and their ecosystems of utility plug-ins. A language is defined as a model in the Ecore format (a meta-modeling language), and starting from the meta-model definition the tool automatically generates model editing facilities as plug-ins for Eclipse. Moreover, there exist many EMF-based MDE tools for defining custom editors, model transformations, and so on.

For the implementation of CAStlE we have defined three metamodels, namely *AdaptiveSystemMM.ecore*, *ensembleModel.ecore*, and *CollectiveAdaptionModel.ecore*. Starting from them and by exploiting the generation facilities of EMF we obtained model editor plug-ins in an automatic way. The checks devoted to validate the conformance of models to their meta-models is also automatically generated by EMF, and is based on the constraints specified in the meta-models themselves (notably multiplicities of the relationships). Moreover, each of the defined meta-models conveys the creation of a corresponding model editor, which allows for the three views of CAStlE to be independently edited and linked together later on. We provide some demonstrative videos showing how to install[21] and run[22] the tool, whose code is available as GIT repository[23].

The first experiences we had using the tool already gave some confidence about the potentials of approaching CAS modeling at a higher level of abstraction as the one offered by CAStlE. The learning phase necessary to a domain expert to be able to use CAStlE is limited to understanding how the different characteristics of a CAS are modeled in the three views. In our case, the domain experts, who were not involved in the concrete implementation of the meta-models, took less than one day to learn the tool and start using it.

From a practical perspective, specifying a CAS as a set of models rather than XML files avoids the typical burdens of dealing with (structured) text

[20] `https://www.eclipse.org/modeling/emf/` [21] `https://youtu.be/EkLBh1RStgQ`
[22] `https://youtu.be/jC76LDCb078` [23] `https://github.com/das-fbk/CAS-DSL`

files. In particular, a XML/text editor is not aware of the CAS-specific concepts and relationships. Therefore, any inconsistency introduced in the specification of the system can only be discovered at a later stage of the development, e.g. by running a check algorithm or when trying to execute the system. It is worth noting that a CAS like the one presented in this dissertation requires thousands of textual lines for its specification. As a consequence, discovering the causes of inconsistencies can be an extremely time-consuming activity. On the contrary, by adopting a DSL as CAStlE the specification is correct-by-construction, in the sense that the tool does not allow the introduction of inconsistencies, otherwise the models would became invalid with respect to their meta-models.

7.5 Discussion

In this chapter, we described the implementation of the design for adaptation framework and its evaluation performed through the execution of ATLAS and DeMOCAS, the demonstrators realized on top of the framework. Each demonstrator allowed us to evaluate different aspects.

ATLAS represents the implementation of the Travel Assistant of our motivating scenario. Its realization has represented the first attempt of wrapping real-world mobility services as domain objects, and dynamically composing them during the system execution. Moreover, the design and execution of ATLAS allowed us to fully cover the round-trip life-cycle of our approach for modeling and executing adaptive by design service-based systems operating in dynamic environments, described in chapter 3. In addition, we recall that the key idea behind ATLAS is that it is open to continuous extensions with new services, wrapped as domain objects. Their functionalities can, thus, be exploited in a transparent way to provide value-added services to the end-users. Indeed, we already emphasized that the more (mobility) services are wrapped up and stored in the system's knowledge base, the more responsive and accurate the travel assistant will be. Currently, we are working on adding new services available as open APIs to extend the application domain of ATLAS. This will also allow us to perform further tests of the system, as made by more domain objects,

that is, more inter-operating services.

Our main concern during the realization of ATLAS, was related to the *feasibility* of our approach. In other words, we were concerning about (i) the time and effort required for wrapping services as domain objects, from the design perspective, and (ii) to what extent the automatic refinement (fragments selection and composition) affects the execution of the system, from the execution perspective. Both of these aspects had a higher priority than the evaluation of other performance values, such as the scalability, since they strictly affect the *effectiveness* and the *applicability* of the whole approach, from the system's modeling phase. As discussed in section 7.2.4, the evaluation results of ATLAS show that the wrapping time of a service as a domain objects ranges from 4 to 6 hours, by considering average complex services and both experienced and non-expert developers. Furthermore, we argue that when we performed these evaluations, CAStlE, the domain specific language (DSL) for defining (collective) adaptive systems, was not yet available. As discussed in section 7.4, this modeling support allows the reduction of the complexity of specifying service-based adaptive systems by providing domain-specific concepts to designers, who do not have to deal with XML files. Indeed, what we expect from our next tests in the near future, where we intend to use CAStlE, is that the wrapping time will decrease compared to its current value.

As regard our second concern about the influence of the automatic refinement on the execution of the system, we performed a comparison between the (average) time required for adaptation and the (average) response time of real-world mobility services, wrapped as domain objects in ATLAS. As discussed in section 7.2.4, we shown that the adaptation responsiveness is equivalent to that of mobility services, thus it does not degrade the system's execution performance.

Eventually, since ATLAS is in continuous evolution, what we plan for the near future is to run new and more extensive real-world battery of tests to confirm, or even improve as we believe, the results obtained so far.

DeMOCAS is a simulator for collective adaptive systems, currently implementing and executing the Urban Mobility System firstly mentioned in chapter 6. Differently from ATLAS that is *single-user* oriented, DeMOCAS simulates collective systems where more than one user are involved.

Moreover, it also implements a collective adaptation algorithm and it deals with the organization of the involved entities in collaborative ensembles.

Our main concern during the realization of DeMOCAS, was related to the *adaptability* and *context-awareness* aspects. In particular, our goal was to demonstrate the *usability* and the *scalability* of the approach when applied to a real-world complex scenario. As discussed in section 7.3.4, the evaluation performed on multiple users and by collecting execution and adaptation statistics show that, considering both less complex and more complex adaptation problems generated by running DeMOCAS, in the worst case they require less than 0.9 seconds to be solved. Of course, the evaluation of the collective adaptation algorithm was also of primarily relevance, and it has been provided and discussed in chapter 6, where the collective approach has been defined.

Eventually, before concluding this discussion and with a complete knowledge of the two versions of our approach (the original and the extended ones), we want to clarify something about the transparency of the adaptation with respect to the users. As detailed in chapter 5, our framework exploits a set of adaptation mechanisms (i.e., refinement mechanism, local adaptation mechanism and compensation mechanism) that can further be combined together by realizing adaptation strategies (i.e., re-refinement strategy, backward adaptation strategy), allowing systems to handle more complex adaptation needs. Each mechanism (and strategy) is triggered by a specific situation. For instance, the refinement mechanism is triggered by the need of executing an abstract activity (i.e., the need of reaching a goal), while the local adaptation mechanism is triggered by the precondition violation of an activity that has to be performed. These situations can arise both as a direct and explicit consequence of the user's actions and as a consequence of unexpected changes in the operational environment in which the user's application is in execution (i.e., because of exogenous events in the operational context). At this point, if we consider both ATLAS and DeMOCAS and how they perform their travel assistance service,it can be noted that when an adaptation need arises, the selection of the adaptation mechanisms and / or strategies to perform is transparent to the user, which will become aware of the effects of the adaptations after their execution. To the contrary, the selection of the proper (mobility) services

used by the mechanisms and / or strategies is indirectly affected by the user through her preferences that she can express when registering to the service (i.e., ATLAS) and even change over time. However, For instance, when refining a *journey planning* abstract activity, the user preferences (e.g., I prefer train journey to ride-share journey) are involved in the selection of the available mobility services whose execution allows the abstract activity goal to be reached. In this specific example, the application will give priority to those mobility solutions served by train services, to the disadvantage of ride-share solutions. Obviously, it can also happen the case in which there are no solutions satisfying the user's preferences, and the provided service composition can represent an undesired outcome for the user. However, this is also strictly related to the availability of services in the system. That is why, as we emphasized through this dissertation, the more (mobility) services are wrapped up and stored in the system's knowledge base, the more responsive and accurate the travel assistant will be. Besides, service providers do their best when defining the adaptation and application logic of their value-added services modeled on top of our platform. They aim to offer high quality services satisfying users as much as possible. For instance, in ATLAS, this is the case in which the travel assistant, by analyzing the inserted data (i.e., departure and destination points), decides between a *global* or a *local* journey planning (modeled as abstract activities), with the idea of providing travel solutions as accurate as possible.

However, the indirect involvement of users in the service selection and composition tasks can be also seen as a limitation of the overall approach. Nevertheless, it can be reduced or overcome in different ways, one of which we have already realized. For instance, one way could be that of considering all the different adaptation solutions that might satisfy a specific adaptation need, if more than one solution are available, and involve the user in the selection of the preferred one. Clearly, this is not trivial at all, but it requires further investigations both from a modeling and a AI planning perspective.

From the viewpoint of our design for adaptation framework, as we mentioned above, we already did some steps towards an increase of the users involvement in the dynamic execution of systems. In particular, in chapter

6 we introduce Collective Adaptive Systems (CAS) and we explained how our approach has been extended with specific constructs and algorithms to model CAS. These extensions, which we are going to recall in the following, besides allowing collective adaptation to be performed, also enable the management of different situations where the involvement of the users is expected and supported. Indeed, in CAS, entities must be able to self-adapt simultaneously and, at the same time, preserve the collaboration and benefits of the system (or sub-system) they are within. Self-adaptation of an individual entity is therefore not only finalized to the achievement of its own respective goals but also to the fulfillment of emerging goals of the dynamically formed sub-systems. Briefly, to allow collective adaptation, first of all the concepts of *role* and *ensemble* have been defined. Each domain object in the system can play a specific role defining how it interacts and collaborates with other entities. Groups of domain objects sharing common goals, instead, are part of the same ensemble. Moreover, domain objects core processes have been extended with *solvers* and *handlers*. While solvers model the ability of a domain object to handle one or more issues, handlers are used to capture issues, during the nominal execution of a domain object, and to trigger the appropriate solver. These extensions to the original approach, required by the need for collective adaptation, allows for adding dynamicity and flexibility to the system models by giving to the developers, among other things, the possibility to foresee and implement a higher interaction with the users and their involvement as proactive users affecting the adaptation results. In particular, this is done in the solvers, which can be defined independently from the involved real services.

To better understand this point, let us consider an example coming from the urban mobility system scenario, given in section 7.3.1. We described the situation of a flexibus trip with three users already on board and two other waiting at different pick-up points on the flexibus route. We then faced the situation in which a road on the route of the flexibus gets interrupted, because of an accident. This forces the flexibus driver to find an alternative route that, however, can be longer that the current one, thus causing to the passengers on board to be late to their destinations. Moreover, it could also result difficult or too expensive in terms of time and money to reach the passengers which are waiting at the pick-up points.

Eventually, we showed as the solution of this (collective) adaptation need, through the exploitation of handlers and solvers, trigger a decision process made by interactions and negotiations among all the involved entities, such as, the flexibus driver, on board and waiting passengers, and the flexibus company. As a consequence, the final adaptation is collectively defined also through a conscious involvement of the affected users.

To sum up, we discussed that, in the original design for adaptation approach, adaptations are transparent to the users, but indirectly affected by their expressed preferences that can also change over time. In the extended version of the approach, instead, the possibility of specifying solvers to deal with specific issues enables and supports a more conscious users participation. In any case, we believe that our approach offers still many open points from which investigate as concern the users involvement (e.g., machine learning techniques can be used to learn from past decisions of users).

In conclusion, we find that the results achieved from the implementation and evaluation of the design for adaptation framework are quite promising, both in terms of the *effectiveness* of the design approach, based on domain objects, and as regard the *scalability* of the exploited adaptation mechanisms. In spite of that, we think that further modeling and implementation efforts are required to improve the whole approach (i.e., by extending it to the application of other adaptation mechanisms) and to extend the application domain on which we run our tests (i.e., by wrapping more services as domain objects), in order to additionally evaluate it.

We refer to chapter 8 for a final overall discussion about the evaluation of the design for adaptation approach against the requirements that we meant to address.

Chapter 8

Conclusion and Future Work

In this dissertation, we presented a solution to the problem of modeling and executing modern service-based systems able to adapt their behavior when operating in *open* and *dynamic* execution environments. We proposed a solution based on a *design for adaptation* approach of service-based adaptive systems (see chapters 4 and 5). Furthermore, the presented approach has been extended in the direction of Collective Adaptive Systems (CAS), to deal with large-scale systems characterized by the collaborative nature of their heterogeneous and autonomous entities, and for which adaptation is a feature of the collectiveness (see chapter 6). To this aim, a collective adapatation algorithm as been implemented. Lastly, both solutions have been implemented and evaluated (see chapter 7).

In summary, in this dissertation we have seen how our design for adaptation approach faces important issues of both modern service-oriented environments and the systems operating within them. In section 3.1 we discussed the requirements that adaptive service-based systems must fulfill. There are commonsensical requirements strictly related to the nature of services (i.e., heterogeneity awareness, autonomy awareness, information accuracy, portability); requirements demanded by modern service ecosystems (i.e., openness awareness, context awareness, adaptivity, interoperability, customizability). Indeed, modern environments are quite challenging and they demand a further effort to shape and execute dynamic and context-dependent services. This effort goes beyond the application of the service-oriented computing set of design principles [36], which already greatly support the development of service-oriented applications. We dis-

cussed in chapter 1 the grand principle behind our approach and how we thought of addressing it, as summarized in the following.

To deal with *open and dynamic environments*, we claim that mechanisms enabling adaptation should be introduced in the life-cycle of systems, both in the *design* and in the *run-time* phases, inspired by the suggestion given in [24]. This enables service-based applications to both *adapt* to the actual context (i.e., the currently available services) and *react* when facing new context situations (e.g., missing services, newly available services, changed services).

To this aim, for an effective management of modern environments, we push the idea of thinking separately about *what* a system must do and *how* it does it. Indeed, while a service-based application can define *what* it does, by specifying its functionalities from its design phase, the concrete implementation of these functionalities, representing *how* they might be effectively accomplished, can vary depending on the execution context, since it can be provided by disparate real-world services, or a composition of them, in a completely dynamic way.

This previous insight strongly suggests keeping separate the *application logic* (that models the *what*) from the *adaptation logic* (that gives the *how*). Indeed, differently from the application logic, the adaptation logic is strongly affected by the run-time operational context of the system, thus it cannot be foreseen when systems are designed. Moreover, without the opportunity to know a priori the involved services and the users' needs, the detachment of the adaptation requirements from the external services specification is crucial.

Most of this dissertation was devoted to show how the presented design for adaptation approach of service-based systems effectively implements the before-mentioned ideas.

The approach mainly relies on two distinct but correlated models: the *Domain model* and the *Domain Objects model*. The former describes the operational environment of the system. Particularly, it allows designers to abstract the *domain concepts* of the referred domain. The latter allows developers to uniformly specify the *autonomous* and *heterogeneous* services in the environment, and their *dynamic interaction*, as the concrete and diverse implementations of the domain concepts specified by the do-

main model. The connection between the two models is achieved through the annotation of the services defined in the domain objects model with *composition requirements* predicating over the domain model. These two correlated models ensure the separation between the application and adaptation logic, and the detachment of adaptation requirements from external services specification. Moreover, the approach allows for the specification of services behaviors (i.e., domain objects core processes) and functionalities (i.e., domain objects fragments) as processes, with the additional possibility of defining *dynamically customizable processes*, that can be concretely specified at run-time, to guarantee the *context-aware execution* of systems. For the context-aware *customization* of processes and the *run-time adaptation* of systems in case of exogenous changes, the approach further exploits automated planning techniques for the *dynamic and incremental service composition* [25], which are particularly suitable in open and dynamic environments. This clearly highlights how the approach implements the separation between *what* a system must do (i.e., customizable processes) and *how* it does it (i.e., customized processes).

Lastly, the implementation of all these features requires both design time constructs (i.e., services wrapped as domain objects, soft dependencies among domain objects, abstract activities, annotations) and run-time mechanisms (i.e., service composition approaches, strong dependencies among domain objects, dynamic domain objects knowledge extension), in line with the intuition in [24], where the authors guessed the importance of introducing mechanisms enabling adaptation both in the *design* and in the *run-time* phases of the life-cycle of systems.

The whole approach (i.e., the models, techniques and related engines) has been implemented in a comprehensive framework on top of which adaptive service-based applications can be modeled and executed. The implementation of two different demonstrators, ATLAS implementing a travel assistant and DeMOCAS enabling the modeling of collective adaptive systems and their execution via a collective adaptation approach, allowed us to evaluate different aspects of the design for adaptation approach, by showing promising results (see chapter 7).

Furthermore, the complete life-cycle of the *continuous development* of adaptive service-based applications within our design for adaptation frame-

work has been described in chapter 3. It gives a complete overview of the different perspectives of the approach (i.e., system models, adaptation, interaction), the involved actors (i.e., platform provider, service providers, end-users) and their participation in the entire life-cycle.

In this chapter, we are going to discuss how the solution developed in this thesis addresses the requirements of service-based systems as listed in section 3.1, as well as the research challenges given in the introduction of this dissertation.

8.1 Requirements Coverage

In section 3.1 we described the motivating example of this thesis work. It is concerned with the management and operation of mobility services, within a smart city as well as among different cities/countries. In particular, it refers to a *Travel Assistant* system, able to support users in the overall organization and execution of their journeys, through the dynamic interoperation of different mobility services.

Moreover, we used the travel assistant scenario to highlight a set of typical *requirements* of modern service-based adaptive systems that require urgent attention to make them able to operate in continuous evolving environments. The aim of this section is that of discussing the coverage provided by our approach to these requirements. In table 8.1 and table 8.2 we recall each requirement and we map it within the design for adaptation framework.

In conclusion, we can say that we addressed the challenges given in chapter 1, beyond some needed improvements that we leave for future work. Indeed, our solution handles all the discussed requirements needed by modern service-based applications as a whole, and not only a subset of them (**RC1** in section 1.2). In addition, through the implementation of the travel assistant described in section 3.1, we demonstrated that our solution can be applied even out of the realm of research (**RC2** in section 1.2).

Of course, we are aware that there are still many extensions and improvements we can consider to make our approach more powerful. In the

Heterogeneity awareness. *The system must consider the heterogeneity of the services in terms of technologies, or in terms of offered functionalities (e.g., REST API vs. SOAP, online booking vs. on board ticketing).* The approach requires the wrapping of different services in a uniform way (i.e., domain objects). This feature enables the possibility of handling in a systematic fashion heterogeneous services, their offered functionalities and the way in which they communicate and relate to each other. Indeed, each domain object models both the service behavior and the service functionalities as processes whose activities implements the effective interaction with the real-world service, in a transparent way. We have seen in chapter 7 that the wrapping time is a matter of hours, by considering average complex services and average experienced developers.
Autonomy awareness. *The system must take into account the autonomous nature of the services involved.* The discussion about the autonomy awareness relates to the heterogeneity awareness. Also in this case, by wrapping services as domain objects we do not prevent their autonomy. The wrapped real-world services and their providers might also be not aware of the domain objects wrapping them, through which they transparently receive service requests. In other words, they continue to autonomously operate in the real-world, while their corresponding domain objects are part of the knowledge base of the adaptive service-based systems developed within our framework.
Openness awareness. *The system must be capable to operate in open environments with continuously entering and leaving services, which are not known a priori (e.g., a new ride-sharing service is available in the city).* In this case, the *domain model* plays an important role. Indeed, as we detailed in chapter 4, the domain model is specified by domain experts and it describes the operational environment of the system. In particular, it is defined as a set of domain properties describing specific concepts of the domain. Then, each real-world service wrapped as domain object in the system must implement one of the domain properties of the domain model. The adaptive execution of systems is, then, driven by rich composition requirements specified on top of the domain model properties, independently of the domain objects implementing them. Thus, when a new service is available in the environment, the only thing that must be done is to define the domain object wrapping it. Indeed, after its wrapping, the service is seamlessly part of the framework and exploited for automatic composition and refinement, without the need of re-deploying the whole system or modifying the system's code in some of its part.
Interoperability. *The system must be capable to propose complex solutions taking advantages of the variety of services (e.g., a user needs of a unique solution with her booking, payment, train journey, and taxi ride).* The core idea is to factorize the capabilities offered by service providers as a set of building blocks (i.e., domain-objects), which can be easily combined with each other through their offered/required functionalities. These relations among domain objects give rise to composite services. Moreover, relevant (composite) services can be published so that stakeholders can personalize and turn them into new available services (i.e. applications, such as the travel assistant). Thanks to this general approach, we facilitate the *integration* and *interoperability* of services that are otherwise independent and autonomous.

Table 8.1: Requirements coverage provided by our solution (Part 1).

Customizability. *The system provide users with personalized solutions (e.g., a wheelchair user has preferences on transportation means and stops).*

The customizability of systems defined within our approach can be managed at least in two different ways. One relies on the same real-world services that are wrapped as domain objects and are part of the systems knowledge base. If they support customization of the service they offer, through their behavior and functionalities, this will be reflected in the domain objects wrapping them (e.g., if a journey planner allows users to specify different preferences before planning, this possibility will be also available in the domain object wrapping the journey planner). The second way, instead, consists in explicitly adding the management of user preferences when implementing domain-specific domain objects (i.e., travel assistant) and then use these preferences to drive the dynamic service selection and composition. While we have the first solution for free, the second one requires of future work. In the current version of our approach we defined a preliminary management of user preferences driving the customizability of systems. However, this requirement asks more attention in the near future extension of our approach.

Information accuracy. *The system must provide up to date and reliable information and solutions (e.g., temporary changes on a bus route).*

The accuracy of the information and solutions provided by the systems is guaranteed by the fulfillment of the *interoperability* and *context awareness* requirements. Indeed, the ability of our approach of enhancing services interoperability allows, not only to exploit their potentialities through their run-time and context-aware discovery and composition, but also to fill their gaps, such as the lack of accuracy. In the mobility domain, for instance, we have discussed in section 7.2.1 the fact that the more global are the services, the more they tend renounce accuracy, contrariwise to local services. Combining the coverage of global services with the accuracy of local services is a concrete example of services interoperability supporting accuracy promoted by our framework, and showed in chapter 7.

Adaptivity and Context awareness. *The system must be able to react and adapt to changes in the environment that might occur and affect its operations (e.g., a strike affects the user's train journey and the system offers a new journey plan), in a context-aware fashion.*

In the presented design for adaptation approach, these requirements are supported from the design of systems to their execution. Their main enablers are (i) the possibility of defining *dynamically customizable processes* when wrapping services behavior and functionalities, and (ii) the exploitation of automated planning techniques, particularly suitable for the execution in open and dynamic environments, for the context-aware customization of processes and the run-time adaptation of systems. In the evaluation of the approach reported in chapter 7, we also discussed the scalability of the adaptation mechanisms exploited within our framework.

Portability. *The system must be deployable in different environments without an ad-hoc reconfiguration from the developers (e.g., the travel assistant must be usable in Trento as well as in Paris).*

The fulfillment of this requirement is particularly guaranteed by the detachment of composition requirements from the external services specification. This is due to the separation of the adaptation and application logic of systems, realized by the domain model and the domain objects model, respectively. Indeed, composition requirements are defined on top of domain properties that abstract specific domain concepts, in a completely separate way from the concrete implementations (i.e., domain objects) of the services implementing these concepts. In this way, if, for instance, the user is organizing a train journey, the execution of the travel assistant relies on the *train journey* property specified in the domain model. Then, depending on the current context of the user (e.g., the city where she is moving) and on the services wrapped up as domain objects in the system, the travel assistant will select and provide to the user the services available for that specific context and situation, without any reconfiguration.

Table 8.2: Requirements coverage provided by our solution (Part 2).

next section we discuss the most important next steps we plan to take in the near future.

8.2 Future Work

Although the solution proposed in this dissertation might overcome some limitations of existing works in the field, there are several open issues we would like to deal with in the near future. We discuss them in the following paragraphs.

Technical improvements.

One of the direction for future work is to further improve the approach, both from the modeling side and the adaptive execution side. Indeed, there are different improvement on which we can work. Among them, one is the possibility of verifying the current state of the external domain knowledge of each domain object through **monitoring facilities** offered by other domain objects (at the moment the state evolves considering only the effect annotations on the received fragments and might not be aligned with the real world situation). Another important extension, related to the previous one, concerns the **support for other forms of run-time adaptation** (e.g., reaction to a context change observed by monitoring the environment). Besides, the current version of our approach deals with functional adaptation mechanisms and strategies. It would be interesting to consider **non-functional service selection** and adaptation strategies (i.e., based on QoS). Furthermore, we plan to **better integrate the data-flow** in our approach. To this aim, we plan to adopt the work in [166] that is compliant with the service composition mechanisms we exploit. We also consider to combine the data and control-flow. For instance, what we plan to do is to relate the definition of an activity precondition not only on the current context state but also on the values of data variables of the context. Going on, the most important improvement probably refer to the **application and evaluation of our solution in distributed environments**. In other words, while up to date we used it in a centralized fashion, we plan to exploit and adapt it for implementing and executing distributed

adaptive systems. This would allow us, for instance, to deal with more than one process engine (i.e., one for each distributed component), by effectively managing the parallel execution of domain objects core processes, while running applications within our framework.

User-centricity in service-oriented systems.

An interesting and relevant topic in the service-oriented computing community is represented by the **user-centricity in service-oriented systems**. Moreover, involving the user in the loop becomes quite more relevant in self-adaptive applications [167]. Modern service-based adaptive systems should adapt to the changing environments considering also the current user intentions, by asking the user himself. Indeed, what can happen is that the self-adaptive behavior of the application, even if fair and relevant with respect to the context, lacks of success if it does not corresponds to the user's intentions. As regard our proposed solution, we believe that the combination of the design for adaptation approach with the service composition mechanisms that the approach exploits, allowed us to take some steps forward in this direction. In particular, our solution already supports the user in performing a variety of different tasks and it deals with reactive behavior. However, we know that this is still not enough, but we expect that our approach to can be extended to user-centric systems. For instance, the approach allows for the specification of abstract composition requirements, defined by domain experts at design time. An extension on which we can work on is the possibility of grounding the abstract composition requirements on services chosen by the user at run-time. A major effort is required, instead, to let the user control the execution of the system via a user-centric service composition, in such a way that she is continuously informed about the execution progress and she can make decisions affecting it.

Internet of Things domains.

As stated in the introduction, the Internet of Services is becoming more and more pervasive, since the trend is to deliver *everything as a service* [1]. Moreover, the scenario is still evolving, with the emergence of

new metaphors, such as that of the Internet of Things (IoT) [168]. An important direction for future work, which we already started working on, concerns with the **application of our solution to IoT domains**. Estimates say that in the coming years billions of devices and objects will form a large heterogeneous and highly distributed system. These devices can be seen as service providers, for instance due to their sensing and actuating capabilities that can be also organized in more complex IoT-based applications to provide context-aware functionalities to the users. Furthermore, also in the context of the IoT, self-adaptation is one of the main concerns. To increase the potentiality of the IoT in improving our ways of living and working, a significant engineering effort is required [169], including the modeling and management of the run-time adaptation of IoT-based systems to different situations. To this aim, we believe that our design for adaptation approach has the potentiality to be easily applied to IoT domains, for modeling and executing IoT-based adaptive systems. This is due, particularly, to the possibility of overcoming the devices heterogeneity by wrapping them as domain objects allowing for a systematic way of managing their interoperability and functionalities composition to specify relevant (composite) IoT-based services. Furthermore, as we already do in the IoS, we might specify abstract requirements that are subsequently refined at run-time by selecting available devices offering the needed functionalities.

Example. *For instance, let us consider meeting rooms equipped with things, i.e., (smart) connected devices and objects such as projectors, speakers, various sensors, chairs, lights, and curtains. These things may provide services such as room temperature sensing, light level sensing, turning on the light, and moving the curtains. In such a scenario, the idea is to dynamically realize a composition of IoT-based functionalities to achieve the goal "deliver presentation". This involves finding a suitable room based on user needs, setting up the display devices and managing dynamic changes in the environment.*

We can notice how this scenario is quite similar to that of the travel assistant in the mobility domain. To consider different scenarios in IoT domains will also enable us to demonstrate that our solution is domain-

independent. This is a relevant characteristic if we consider the evolution of the IoS in the panorama of the Next Generation Internet.

Services evolution.

As known, adaptation may be divided into *short-term adaptation* and **long-term adaptation**, also know as *evolution*. Differently from the former, the evolution implies definitive changes to the system that will affect its future operation. Thinking about services, evolution stands for changes to their behavior and exposed functionalities (i.e., domain objects core processes and fragments) so that they will be propagated to all their future instances. In this direction, we plan to adopt and further extend the work in [34], where we started to reason about a service co-evolution approach. In particular, in [34] we have presented a solution for service co-evolution, based on the Domain Object concept, which supports deep changes across a service dependency graph, through the decentralized collaboration of evolution agents. Based on a classification of externally noticeable service changes and their potential implications on dependent services, we have discussed how our approach can automate several types of changes at run time, and facilitates the coordination of manual maintenance activities in a number of others. However, although our contribution is a significant step towards on-the-fly service evolution, we are not there yet.

Bibliography

[1] George Pallis. Cloud computing: The new frontier of internet computing. *IEEE Internet Computing*, 14(5):70–73, 2010.

[2] Rafael Moreno-Vozmediano, Rubén S. Montero, and Ignacio Martín Llorente. Key challenges in cloud computing: Enabling the future internet of services. *IEEE Internet Computing*, 17(4):18–25, 2013.

[3] European Commission. Next generation internet initiative, 2016. `https://ec.europa.eu/digital-single-market/en/policies/next-generation-internet`.

[4] Cross-European Technology Platforms (X-ETPs) Group. Future internet strategic research agenda, ver. 1.1, 2010. `http://www.future-internet.eu/news/view/article/x-etp-group-future-internet-strategic-research-agenda.html`.

[5] Luciano Baresi, Elisabetta Di Nitto, and Carlo Ghezzi. Toward open-world software: Issue and challenges. *IEEE Computer*, 39(10):36–43, 2006.

[6] COTEVOS – Concepts, Capacities and Methods for Testing EV systems and their interOperability within the Smartgrids. FP7 ENERGY WorkProgramme – Call ENERGY.2013.7.3.2 - Enhanced interoperability and conformance testing methods and tools for interaction between grid infrastructure and electric vehicles, 2013-2016. `http://cotevos.eu/`.

[7] MoveUs: ICT cloud-based platform and mobility services: available, universal and safe for all users. FP7 ICT WorkProgramme – Call ICT-2013.6.6 Integrated personal mobility for smart cities, 2013-2016. `http://www.moveus-project.eu/`.

[8] STREETLIFE – Steering Towards Green and Perceptive Mobility of the Future. FP7 ICT WorkProgramme – Call ICT-2013.6.6 Integrated personal mobility for smart cities, 2013-2016. `http://www.streetlife-project.eu`.

[9] Marco Pistore, Paolo Traverso, Massimo Paolucci, and Matthias Wagner. From software services to a future internet of services. In *Towards the Future Internet - A European Research Perspective*, pages 183–192, 2009.

[10] Athman Bouguettaya, Munindar P. Singh, Michael N. Huhns, Quan Z. Sheng, Hai Dong, Qi Yu, Azadeh Ghari Neiat, Sajib Mistry, Boualem Benatallah, Brahim Medjahed, Mourad Ouzzani, Fabio Casati, Xumin Liu, Hongbing Wang, Dimitrios Georgakopoulos, Liang Chen, Surya Nepal, Zaki Malik, Abdelkarim Erradi, Yan Wang, M. Brian Blake, Schahram Dustdar, Frank Leymann, and Michael P. Papazoglou. A service computing manifesto: the next 10 years. *Commun. ACM*, 60(4):64–72, 2017.

[11] Valérie Issarny, Nikolaos Georgantas, Sara Hachem, Apostolos V. Zarras, Panos Vassiliadis, Marco Autili, Marco Aurélio Gerosa, and Amira Ben Hamida. Service-oriented middleware for the future internet: state of the art and research directions. *J. Internet Services and Applications*, 2(1):23–45, 2011.

[12] James Lewis and Martin Fowler. Microservices in a nutshell, 2014.

[13] Sam Newman. *Building Microservices – Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[14] Gunnar Brataas, Svein O. Hallsteinsen, Romain Rouvoy, and Frank Eliassen. Scalability of decision models for dynamic product lines. In *Software Product Lines, 11th International Conference, SPLC 2007, Kyoto, Japan, September 10-14, 2007, Proceedings. Second Volume (Workshops)*, pages 23–32, 2007.

[15] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[16] Christoph Seidl, Sven Schuster, and Ina Schaefer. Generative software product line development using variability-aware design patterns. *Computer Languages, Systems & Structures*, 48:89–111, 2017.

[17] Christoph Dorn, Daniel Schall, and Schahram Dustdar. Context-aware adaptive service mashups. In *4th IEEE Asia-Pacific Services Computing Conference, IEEE APSCC 2009, Proceedings*, pages 301–306, 2009.

[18] Jian Yu, Quan Z. Sheng, and Joshua K. Y. Swee. Model-driven development of adaptive service-based systems with aspects and rules. In *Web Information Systems Engineering - WISE 2010 - 11th International Conference, Hong Kong, China, December 12-14, 2010. Proceedings*, pages 548–563, 2010.

[19] Richard Hull, Elio Damaggio, Riccardo De Masellis, Fabiana Fournier, Manmohan Gupta, Fenno F. Terry Heath III, Stacy Hobson, Mark H. Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Noi Sukaviriya, and Roman Vaculín. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *Proceedings of the Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011, New York, NY, USA, July 11-15, 2011*, pages 51–62, 2011.

[20] Svein O. Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic software product lines. In *Systems and Software Variability Management, Concepts, Tools and Experiences*, pages 253–260. 2013.

[21] Nour Assy, Nguyen Ngoc Chan, and Walid Gaaloul. An automated approach for assisting the design of configurable process models. *IEEE Trans. Services Computing*, 8(6):874–888, 2015.

[22] Ahmed Tealeb, Ahmed Awad, and Galal H. Galal-Edeen. Context-based variant generation of business process models. In *Enterprise, Business-Process and Information Systems Modeling - 15th International Conference, BPMDS 2014, 19th International Conference, EMMSAD 2014, Held at CAiSE 2014, Thessaloniki, Greece, June 16-17, 2014. Proceedings*, pages 363–377.

[23] Fundamentals of Collective Adaptive Systems (FoCAS). A roadmap to the future of Collective Adaptive Systems, 2016. `http://www.focas.eu/manifesto/`.

[24] Antonio Bucchiarone, Cinzia Cappiello, Elisabetta Di Nitto, Raman Kazhamiakin, Valentina Mazza, and Marco Pistore. Design for adaptation of service-based applications: Main issues and requirements. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops - International Workshops, ICSOC/ServiceWave 2009, Stockholm, Sweden, November 23-27, 2009, Revised Selected Papers*, pages 467–476, 2009.

[25] Antonio Bucchiarone, Annapaola Marconi, Marco Pistore, and Heorhi Raik. A context-aware framework for dynamic composition of process fragments in the internet of services. *J. Internet Services and Applications*, 8(1):6:1–6:23, 2017.

[26] Antonio Bucchiarone, Martina De Sanctis, and Marco Pistore. Domain objects for dynamic and incremental service composition. In *Service-Oriented and Cloud Computing - Third European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings*, pages 62–80, 2014.

[27] Antonio Bucchiarone, Martina De Sanctis, Annapaola Marconi, Marco Pistore, and Paolo Traverso. Design for adaptation of distributed service-based systems. In *Service-Oriented Computing - 13th International Conference, ICSOC 2015, Goa, India, November 16-19, 2015, Proceedings*, pages 383–393, 2015.

[28] Antonio Bucchiarone, Martina De Sanctis, Annapaola Marconi, Marco Pistore, and Paolo Traverso. Incremental composition for adaptive by-design service based systems. In *IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 236–243, 2016.

[29] Antonio Bucchiarone, Martina De Sanctis, and Annapaola Marconi. ATLAS: A world-wide travel assistant exploiting service-based adaptive technologies. In *Service-Oriented Computing - 15th International Conference, ICSOC 2017, Malaga, Spain, November 13-16, 2017, Proceedings*, pages 561–570, 2017.

[30] Antonio Bucchiarone, Martina De Sanctis, and Annapaola Marconi. Decentralized dynamic adaptation for service-based collective adaptive systems. In *Service-Oriented Computing - ICSOC 2016 Workshops - ASOCA, ISyCC, BSCI, and Satellite Events, Banff, AB, Canada, October 10-13, 2016. Revised Selected Papers*, pages 5–20, 2016.

[31] Antonio Bucchiarone, Martina De Sanctis, Annapaola Marconi, and Alberto Martinelli. Democas: Domain objects for service-based collective adaptive systems. In *Service-Oriented Computing - ICSOC 2016 Workshops - ASOCA, ISyCC, BSCI, and Satellite Events, Banff, AB, Canada, October 10-13, 2016. Revised Selected Papers*, pages 174–178, 2016.

[32] Antonio Bucchiarone, Antonio Cicchetti, and Martina De Sanctis. Towards a domain specific language for engineering collective adaptive systems. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 19–26, 2017.

[33] Antonio Bucchiarone, Antonio Cicchetti, and Martina De Sanctis. Castle: A tool for collective adaptive systems engineering. In *2nd IEEE International Workshops on Foundations and Applications of Self* Systems, FAS*W@SASO/ICCAC 2017, Tucson, AZ, USA, September 18-22, 2017*, pages 385–386, 2017.

[34] Martina De Sanctis, Kurt Geihs, Antonio Bucchiarone, Giuseppe Valetto, Annapaola Marconi, and Marco Pistore. Distributed service co-evolution based on domain objects. In *Service-Oriented Computing - ICSOC 2015 Workshops - WESOA, RMSOC, ISC, DISCO, WESE, BSCI, FOR-MOVES, Goa, India, November 16-19, 2015, Revised Selected Papers*, pages 48–63, 2015.

[35] ALLOW Ensembles. FP7 ICT-2011.9.10 - FET Proactive: Fundamentals of Collective Adaptive Systems (FOCAS), 2013-2016. `http://www.allow-ensembles.eu`.

[36] Thomas Erl. *SOA Principles of Service Design*. The Prentice Hall Service Oriented Computing Series. Prentice Hall, 2007.

[37] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly Media, 1st edition, 2007.

[38] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer, 2004.

[39] W3C. Web services glossary, 2004. `http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/`.

[40] Quan Z. Sheng, Xiaoqiang Qiao, Athanasios V. Vasilakos, Claudia Szabo, Scott Bourne, and Xiaofei Xu. Web services composition: A decade's overview. *Inf. Sci.*, 280:218–238, 2014.

[41] Fabio Casati, Ski Ilnicki, Li-jie Jin, Vasudev Krishnamoorthy, and Ming-Chien Shan. Adaptive and dynamic service composition in *eFlow*. In *Advanced Information Systems Engineering, 12th International Conference CAiSE 2000, Stockholm, Sweden, June 5-9, 2000, Proceedings*, pages 13–31, 2000.

[42] Daniela Berardi, Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Massimo Mecella. Automatic composition of e-services that export their behavior. In *Service-Oriented Computing - ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003, Proceedings*, pages 43–58, 2003.

[43] Cesare Pautasso and Gustavo Alonso. Flexible binding for reusable composition of web services. In *Software Composition, 4th International Workshop, SC 2005, Edinburgh, UK, April 9, 2005, Revised Selected Papers*, pages 151–166, 2005.

[44] Chris Peltz. Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52, 2003.

[45] Bpel4people. In *Encyclopedia of Social Network Analysis and Mining*, page 82. 2014.

[46] George Baryannis, Olha Danylevych, Dimka Karastoyanova, Kyriakos Kritikos, Philipp Leitner, Florian Rosenberg, and Branimir Wetzstein. Service composition. In *Service Research Challenges and Solutions for the Future Internet - S-Cube - Towards Engineering, Managing and Adapting Service-Based Systems*, pages 55–84. 2010.

[47] Piergiorgio Bertoli, Marco Pistore, and Paolo Traverso. Automated composition of web services via planning in asynchronous domains. *Artif. Intell.*, 174(3-4):316–361, 2010.

[48] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service-oriented design and development methodology. *Int. J. Web Eng. Technol.*, 2(4):412–442, 2006.

[49] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: a research roadmap. *Int. J. Cooperative Inf. Syst.*, 17(2):223–255, 2008.

[50] Barbara Pernici. Methodologies for design of service-based systems. In *Intentional Perspectives on Information Systems Engineering.*, pages 307–318. 2010.

[51] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[52] Eric Schmidt. Conversation with eric schmidt hosted by danny sullivan, 2006. Search Engine Strategies Conference `https://www.google.com/press/podium/ses2006.html`.

[53] Cesare Pautasso, Thomas Heinis, and Gustavo Alonso. Autonomic resource provisioning for software business processes. *Information & Software Technology*, 49(1):65–80, 2007.

[54] Mehdi Bahrami and Mukesh Singhal. DCCSOA: A dynamic cloud computing service-oriented architecture. In *2015 IEEE International Conference on Information Reuse and Integration, IRI 2015, San Francisco, CA, USA, August 13-15, 2015*, pages 158–165, 2015.

[55] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *CoRR*, abs/1606.04036, 2016.

[56] Olaf Zimmermann. Microservices tenets. *Computer Science - R&D*, 32(3-4):301–310, 2017.

[57] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98, 2017.

[58] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods - Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 194–210, 2016.

[59] Fabrizio Montesi, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. JOLIE: a java orchestration language interpreter engine. *Electr. Notes Theor. Comput. Sci.*, 181:19–33, 2007.

[60] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In *Web Services Foundations*, pages 81–107. 2014.

[61] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 211–222, 2014.

[62] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(2):97–104, 2017.

[63] Kizilov Mikhail, Antonio Bucchiarone, Manuel Mazzara, Larisa Safina, and Víctor Rivera. Domain objects and microservices for systems development: a roadmap. *CoRR*, abs/1709.10255, 2017.

[64] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 1st edition, 2015.

[65] Jaejoon Lee and Gerald Kotonya. Combining service-orientation with product line engineering. *IEEE Software*, 27(3):35–41, 2010.

[66] Charles W. Krueger. Variation management for software production lines. In *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, pages 37–48, 2002.

[67] Vasilios Tzeremes and Hassan Gomaa. A software product line approach for end user development of smart spaces. In *5th IEEE/ACM International Workshop on Product Line Approaches in Software Engineering, PLEASE 2015, Florence, Italy, May 19, 2015*, pages 23–26, 2015.

[68] Luca Gherardi, Dominique Hunziker, and Gajamohan Mohanarajah. A software product line approach for configuring cloud robotics applications. In *2014 IEEE 7th International Conference on Cloud Computing, Anchorage, AK, USA, June 27 - July 2, 2014*, pages 745–752, 2014.

[69] Ivonei Freitas da Silva, Paulo Anselmo da Mota Silveira Neto, Pádraig O'Leary, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. Using a multimethod approach to understand agile software product lines. *Information & Software Technology*, 57:527–542, 2015.

[70] Reto Krummenacher, Barry Norton, and Adrian Marte. Towards linked open services and processes. In *Future Internet - FIS 2010 - Third Future Internet Symposium, Berlin, Germany, September 20-22, 2010. Proceedings*, pages 68–77, 2010.

[71] Meherun Nesa Lucky, Marco Cremaschi, Barbara Lodigiani, Antonio Menolascina, and Flavio De Paoli. Enriching API descriptions by adding API profiles through semantic annotation. In *Service-Oriented Computing - 14th International Conference, ICSOC 2016, Banff, AB, Canada, October 10-13, 2016, Proceedings*, pages 780–794, 2016.

[72] Moshe Chai Barukh and Boualem Benatallah. Servicebase: A programming knowledge-base for service oriented development. In *Database Systems for Advanced Applications, 18th International Conference, DASFAA 2013, Wuhan, China, April 22-25, 2013. Proceedings, Part II*, pages 123–138, 2013.

[73] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23-25, 2007, Minneapolis, MN, USA*, pages 259–268, 2007.

[74] Danny Weyns and Michael P. Georgeff. Self-adaptation using multiagent systems. *IEEE Software*, 27(1):86–91, 2010.

[75] Gary Look, Stephen Peters, and Howard Shrobe. Plan-driven ubiquitous computing. In *UbiComp*, 2006.

[76] J. P. Sousa, V. Poladian, D. Garlan, B. Schmerl, and M. Shaw. Task-based adaptation for ubiquitous computing. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36(3):328–340, 2006.

[77] Elisabetta Di Nitto, Carlo Ghezzi, Andreas Metzger, Mike P. Papazoglou, and Klaus Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Autom. Softw. Eng.*, 15(3-4):313–341, 2008.

[78] Walter Binder, Daniele Bonetta, Cesare Pautasso, Achille Peternier, Diego Milano, Heiko Schuldt, Nenad Stojnic, Boi Faltings, and Immanuel Trummer. Towards self-organizing service-oriented architectures. In *World Congress on Services, SERVICES 2011, Washington, DC, USA, July 4-9, 2011*, pages 115–121, 2011.

[79] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[80] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. *Software Engineering for Self-Adaptive Systems: A Second Research Roadmap*, pages 1–32. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[81] Jörg Hoffmann, Ingo Weber, and Frank Michael Kraft. SAP speaks PDDL: exploiting a software-engineering model for planning in business process management. *J. Artif. Intell. Res.*, 44:587–632, 2012.

[82] Mahmoud Hussein, Jun Han, Jian Yu, and Alan Colman. Enabling runtime evolution of context-aware adaptive services. In *2013 IEEE International Conference on Services Computing, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 248–255, 2013.

[83] Brice Morin, Olivier Barais, Jean-Marc Jézéquel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *IEEE Computer*, 42(10):44–51, 2009.

[84] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *IEEE Computer*, 42(10):22–27, 2009.

[85] Adina D. Mosincat, Walter Binder, and Mehdi Jazayeri. Runtime adaptability through automated model evolution. In *Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, EDOC 2010, Vitória, Brazil, 25-29 October 2010*, pages 217–226, 2010.

[86] Sira Yongchareon, Chengfei Liu, and Xiaohui Zhao. A framework for behavior-consistent specialization of artifact-centric business processes. In *Business Process Management - 10th International Conference, BPM 2012, Tallinn, Estonia, September 3-6, 2012. Proceedings*, pages 285–301, 2012.

[87] Javier Cubo, Nadia Gámez, Lidia Fuentes, and Ernesto Pimentel. Composition and self-adaptation of service-based systems with feature models. In *Safe and Secure Software Reuse - 13th International Conference on Software Reuse, ICSR 2013, Pisa, Italy, June 18-20. Proceedings*, pages 326–342, 2013.

[88] Aitor Murguzur, Salvador Trujillo, Hong Linh Truong, Schahram Dustdar, Óscar Ortiz, and Goiuria Sagardui. Run-time variability for context-aware smart workflows. *IEEE Software*, 32(3):52–60, 2015.

[89] Javier Cubo and Ernesto Pimentel. Damasco: A framework for the automatic composition of component-based and service-oriented architectures. In *Software Architecture - 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings*, pages 388–404, 2011.

[90] Kevin Göser, Martin Jurisch, Hilmar Acker, Ulrich Kreher, Markus Lauer, Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Next-generation process management with ADEPT2. In *Proceedings of the BPM Demonstration Program at the Fifth International Conference on Business Process Management (BPM'07), Brisbane, Australia, 24-27 September 2007*, 2007.

[91] Michael Rosemann and Wil M. P. van der Aalst. A configurable reference modelling language. *Inf. Syst.*, 32(1):1–23, 2007.

[92] Mahdi Bashari, Ebrahim Bagheri, and Weichang Du. Automated composition of service mashups through software product line engineering. In *Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings*, pages 20–38, 2016.

[93] Andrei Ciortea, Olivier Boissier, Antoine Zimmermann, and Adina Magda Florea. Responsive decentralized composition of service mashups for the internet of things. In *Proceedings of the 6th International Conference on the Internet of Things, IOT 2016, Stuttgart, Germany, November 7-9, 2016*, pages 53–61, 2016.

[94] Nam Ky Giang, Michael Blackstock, Rodger Lea, and Victor C. M. Leung. Developing iot applications in the fog: A distributed dataflow approach. In *5th International Conference on the Internet of Things, IOT 2015, Seoul, South Korea, 26-28 October, 2015*, pages 155–162, 2015.

[95] Meir M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. In *Proceedings of the IEEE*, volume 68, pages 1060–1076, September 1980.

[96] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Software Eng.*, 16(11):1293–1306, 1990.

[97] Mike P. Papazoglou. The challenges of service evolution. In *CAiSE*, volume 5074 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2008.

[98] Vasilios Andrikopoulos, Salima Benbernou, and Michael P. Papazoglou. On the evolution of services. *IEEE Trans. Software Eng.*, 38(3):609–628, 2012.

[99] Seung Hwan Ryu, Fabio Casati, Halvard Skogsrud, Boualem Benatallah, and Régis Saint-Paul. Supporting the dynamic evolution of web service protocols in service-oriented architectures. *TWEB*, 2(2):13:1–13:46, 2008.

[100] David Webster, Paul Townend, and Jie Xu. Restructuring web service interfaces to support evolution. In *8th IEEE International Symposium on Service Oriented System Engineering, SOSE 2014, Oxford, United Kingdom, April 7-11, 2014*, pages 158–159, 2014.

[101] Daniele Romano and Martin Pinzger. Analyzing the evolution of web services using fine-grained changes. In *2012 IEEE 19th International Conference on Web Services, Honolulu, HI, USA, June 24-29, 2012*, pages 392–399, 2012.

[102] Marios Fokaefs and Eleni Stroulia. Wsdarwin: Studying the evolution of web service systems. In *Advanced Web Services*, pages 199–223. 2014.

[103] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[104] Svein O. Hallsteinsen, Kurt Geihs, Nearchos Paspallis, Frank Eliassen, Geir Horn, Jorge Lorenzo, Alessandro Mamelli, and George Angelos Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *Journal of Systems and Software*, 85(12):2840–2859, 2012.

[105] Kurt Geihs, Christoph Evers, Roland Reichle, Michael Wagner, and Mohammad Ullah Khan. Development support for qos-aware service-adaptation in ubiquitous computing applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 197–202, 2011.

[106] Vincent Marchau, Warren Walker, and Ron van Duin. An adaptive approach to implementing innovative urban transport solutions. *Transport Policy*, 15(6):405 – 412, 2008.

[107] Sridevi Saralaya and Rio D'Souza. A review of monitoring techniques for service based applications. In *2013 2nd International Conference on Advanced Computing, Networking and Security, Mangalore, India, December 15-17, 2013*, pages 96–101, 2013.

[108] Hatim Guermah, Tarik Fissaa, Hatim Hafiddi, Mahmoud Nassar, and Abdelaziz Kriouile. Context modeling and reasoning for building context aware services. In *ACS International Conference on Computer Systems and Applications, AICCSA 2013, Ifrane, Morocco, May 27-30, 2013*, pages 1–7, 2013.

[109] E. Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.

[110] Hanna Eberle, Tobias Unger, and Frank Leymann. Process fragments. In *On the Move to Meaningful Internet Systems: OTM 2009, Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009, Vilamoura, Portugal, November 1-6, 2009, Proceedings, Part I*, pages 398–405, 2009.

[111] Mark Deck and Mark Strom. Model of co-development emerges. *Research-Technology Management*, 45(3):47–53, 2002.

[112] Florence Crespin-Mazet and Pervez Ghauri. Co-development as a marketing strategy in the construction industry. *Industrial Marketing Management*, 36(2):158–172, 2007.

[113] Enrique Estellés-Arolas and Fernando González-Ladrón-de Guevara. Towards an integrated crowdsourcing definition. *Journal of Information science*, 38(2):189–200, 2012.

[114] Jeff Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.

[115] Paul Duvall, Stephen M. Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series).* Addison-Wesley Professional, 2007.

[116] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 1st edition, 2010.

[117] Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M. Verner, and Markku Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, 123:263–291, 2017.

[118] Piergiorgio Bertoli, Raman Kazhamiakin, Massimo Paolucci, Marco Pistore, Heorhi Raik, and Matthias Wagner. Control flow requirements for automated service composition. In *IEEE International Conference on Web Services, ICWS 2009, Los Angeles, CA, USA, 6-10 July 2009*, pages 17–24, 2009.

[119] Piergiorgio Bertoli and Marco Pistore. Planning with extended goals and partial observability. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004), June 3-7 2004, Whistler, British Columbia, Canada*, pages 270–278, 2004.

[120] Annapaola Marconi, Marco Pistore, and Paolo Traverso. Automated composition of web services: the ASTRO approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.

[121] Heorhi Raik, Antonio Bucchiarone, Nawaz Khurshid, Annapaola Marconi, and Marco Pistore. Astro-captevo: Dynamic context-aware adaptation for service-based systems. In *Eighth IEEE World Congress on Services, SERVICES 2012, Honolulu, HI, USA, June 24-29, 2012*, pages 385–392, 2012.

[122] Adina Sirbu, Annapaola Marconi, Marco Pistore, Hanna Eberle, Frank Leymann, and Tobias Unger. Dynamic composition of pervasive process fragments. In *IEEE International Conference on Web Services, ICWS 2011, Washington, DC, USA, July 4-9, 2011*, pages 73–80, 2011.

[123] Antonio Bucchiarone, Alberto Lluch-Lafuente, Annapaola Marconi, and Marco Pistore. A formalisation of adaptable pervasive flows. In *Web Services and Formal Methods, 6th International Workshop, WS-FM 2009, Bologna, Italy, September 4-5, 2009, Revised Selected Papers*, pages 61–75, 2009.

[124] Annapaola Marconi, Marco Pistore, and Paolo Traverso. Specifying data-flow requirements for the automated composition of web services. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*, pages 147–156, 2006.

[125] Marco Pistore, Annapaola Marconi, Piergiorgio Bertoli, and Paolo Traverso. Automated composition of web services by planning at the knowledge level. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1252–1259, 2005.

[126] Rick Rabiser, Sam Guinea, Michael Vierhauser, Luciano Baresi, and Paul Grünbacher. A comparison framework for runtime monitoring approaches. *Journal of Systems and Software*, 125:309–321, 2017.

[127] Antonio Bucchiarone, Annapaola Marconi, Marco Pistore, and Heorhi Raik. Dynamic adaptation of fragment-based and context-aware business processes. In *2012 IEEE 19th International Conference on Web Services, Honolulu, HI, USA, June 24-29, 2012*, pages 33–41, 2012.

[128] Antonio Bucchiarone, Annapaola Marconi, Claudio Antares Mezzina, Marco Pistore, and Heorhi Raik. On-the-fly adaptation of dynamic service-based systems: Incrementality, reduction and reuse. In *Service-Oriented Computing - 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*, pages 146–161, 2013.

[129] Franco Zambonelli, Nicola Bicocchi, Giacomo Cabri, Letizia Leonardi, and Mariachiara Puviani. On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In *Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems, SASOW 2011, Ann Arbor, MI, USA, October 3-7, 2011, Workshops Proceedings*, pages 108–113, 2011.

[130] Philippe Lalanda, Julie A. McCann, and Ada Diaconescu. *Autonomic Computing - Principles, Design and Implementation*. Undergraduate Topics in Computer Science. Springer, 2013.

[131] Danny Weyns and Jesper Andersson. On the challenges of self-adaptation in systems of systems. In *Proceedings of the First International Workshop on Software Engineering for Systems-of-Systems, SESoS@ECOOP, Montpellier, France, July 2, 2013*, pages 47–51, 2013.

[132] Bryan Horling and Victor R. Lesser. A survey of multi-agent organizational paradigms. *Knowledge Eng. Review*, 19(4):281–316, 2004.

[133] Tomasz P. Michalak, Jacek Sroka, Talal Rahwan, Michael Wooldridge, Peter McBurney, and Nicholas R. Jennings. A distributed algorithm for anytime coalition structure generation. In *9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), Toronto, Canada, May 10-14, 2010, Volume 1-3*, pages 1007–1014, 2010.

[134] Sarvapali D. Ramchurn, Maria Polukarov, Alessandro Farinelli, Cuong Truong, and Nicholas R. Jennings. Coalition formation with spatial and temporal constraints. In Wiebe van der Hoek, Gal A. Kaminka, Yves Lesprance, Michael Luck, and Sandip Sen, editors, *AAMAS*, pages 1181–1188. IFAAMAS, 2010.

[135] Jonathan Bonnet, Marie Pierre Gleizes, Elsy Kaddoum, Serge Rainjonneau, and Grégory Flandin. Multi-satellite mission planning using a self-adaptive multi-agent system. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, September 21-25, 2015*, pages 11–20, 2015.

[136] Dayong Ye, Minjie Zhang, and Danny Sutanto. Self-adaptation-based dynamic coalition formation in a distributed agent network: A mechanism and a brief survey. *IEEE Trans. Parallel Distrib. Syst.*, 24(5):1042–1051, 2013.

[137] Hyung Jun Ahn, Habin Lee, and Sung Joo Park. A flexible agent system for change adaptation in supply chains. *Expert Syst. Appl.*, 25(4):603–618, 2003.

[138] Nicholas R. Jennings. An agent-based approach for building complex software systems. *Commun. ACM*, 44(4):35–41, 2001.

[139] Radu-Casian Mihailescu, Matteo Vasirani, and Sascha Ossowski. Dynamic coalition adaptation for efficient agent-based virtual power plants. In *Multiagent System Technologies - 9th German Conference, MATES 2011, Berlin, Germany, October 6-7, 2011. Proceedings*, volume 6973 of *Lecture Notes in Computer Science*, pages 101–112. Springer, 2011.

[140] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Developing correct, distributed, adaptive software. *Sci. Comput. Program.*, 97:41–46, 2015.

[141] Rocco De Nicola, Michele Loreti, Rosario Pugliese, and Francesco Tiezzi. A formal approach to autonomic systems programming: The SCEL language. *TAAS*, 9(2):7:1–7:29, 2014.

[142] Rolf Hennicker and Annabelle Klarl. Foundations for ensemble modeling - the helena approach - handling massively distributed systems with elaborate ensemble architectures. In *Specification, Algebra, and Software - Essays Dedicated to Kokichi Futatsugi*, pages 359–381, 2014.

[143] George Mathews, Hugh Durrant-Whyte, and Mikhail Prokopenko. Decentralized Decision Making for Multiagent Systems. In Mikhail Prokopenko, editor, *Advances in Applied Self-organizing Systems*, Advanced Information and Knowledge Processing, pages 77–104. Springer London, 2008.

[144] Tomonari Furukawa, Hugh F. Durrant-Whyte, and Gamini Dissanayake. Time-optimal cooperative control of multiple robot vehicles. In *Proceedings of the 2003 IEEE International Conference on Robotics and Automation, ICRA 2003, September 14-19, 2003, Taipei, Taiwan*, pages 944–950, 2003.

[145] Chih-Han Yu and Radhika Nagpal. Self-adapting modular robotics: A generalized distributed consensus framework. In *2009 IEEE International Conference on Robotics and Automation, ICRA 2009, Kobe, Japan, May 12-17, 2009*, pages 1881–1888, 2009.

[146] Danny Weyns, Robrecht Haesevoets, and Alexander Helleboogh. The MACODO organization model for context-driven dynamic agent organizations. *TAAS*, 5(4):16:1–16:29, 2010.

[147] Leonardo A. F. Leite, Gustavo Ansaldi Oliva, Guilherme M. Nogueira, Marco Aurélio Gerosa, Fabio Kon, and Dejan S. Milojicic. A systematic literature review of service choreography adaptation. *Service Oriented Computing and Applications*, 7(3):199–216, 2013.

[148] Tomás Bures, Ilias Gerostathopoulos, Petr Hnetynka, Jaroslav Keznikl, Michal Kit, and Frantisek Plasil. DEECO: an ensemble-based component system. In *CBSE'13, Proceedings of the 16th ACM SIGSOFT Symposium on Component Based Software Engineering, part of Comparch '13, Vancouver, BC, Canada, June 17-21, 2013*, pages 81–90, 2013.

[149] Stefan Niemczyk and Kurt Geihs. Adaptive run-time models for groups of autonomous robots. In *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, pages 127–133, 2015.

[150] Christopher Zhong and Scott A. DeLoach. Runtime models for automatic reorganization of multi-robot systems. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*, pages 20–29, 2011.

[151] Danny Weyns, Sam Malek, and Jesper Andersson. On decentralized self-adaptation: lessons from the trenches and challenges for the future. In *2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, Cape Town, South Africa, May 3-4, 2010*, pages 84–93, 2010.

[152] Pieter Vromant, Danny Weyns, Sam Malek, and Jesper Andersson. On interacting control loops in self-adaptive systems. In *2011 ICSE Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, Waikiki, Honolulu , HI, USA, May 23-24, 2011*, pages 202–207, 2011.

[153] Radu Calinescu, Simos Gerasimou, and Alec Banks. Self-adaptive Software with Decentralised Control Loops. In *Fundamental Approaches to Software Engineering - 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 235–251, 2015.

[154] Paul Levi and Serge Kernbach. *Symbiotic Multi-Robot Organisms - Reliability, Adaptability, Evolution*, volume 7 of *Cognitive Systems Monographs*. Springer, 2010.

[155] Behrouz Homayoun Far, Tom Wanyama, and Sidi O. Soueina. A negotiation model for large scale multi-agent systems. In *Proceedings of the 2006 IEEE International Conference on Information Reuse and Integration, IRI - 2006: Heuristic Systems Engineering, September 16-18, 2006, Waikoloa, Hawaii, USA*, pages 589–594, 2006.

[156] Darko Bozhinoski, Ivano Malavolta, Antonio Bucchiarone, and Annapaola Marconi. Sustainable safety in mobile multi-robot systems via collective adaptation. In *2015 IEEE 9th International Conference on Self-Adaptive and Self-Organizing Systems, Cambridge, MA, USA, September 21-25, 2015*, pages 172–173, 2015.

[157] IBM. An architectural blueprint for autonomic computing. Technical report, IBM, 2006.

[158] Thomas L Saaty. *What is the analytic hierarchy process?* Springer, 1988.

[159] Antonio Bucchiarone, Naranker Dulay, Anna Lavygina, Annapaola Marconi, Heorhi Raik, and Alessandra Russo. An approach for collective adaptation in socio-technical systems. In *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops 2015, Cambridge, MA, USA, September 21-25, 2015*, pages 43–48, 2015.

[160] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering*. Springer, 2012.

[161] Antonio Bucchiarone, Naranker Dulay, Anna Lavygina, Annapaola Marconi, Heorhi Raik, and Alessandra Russo. An approach for collective adaptation in socio-technical systems. In *2015 IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, SASO Workshops 2015, Cambridge, MA, USA, September 21-25, 2015*, pages 43–48, 2015.

[162] Darko Bozhinoski, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Massimo Tivoli. FLYAQ: enabling non-expert users to specify and generate missions of autonomous multicopters. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 801–806, 2015.

[163] Jörg Lenhard, Vincenzo Ferme, Simon Harrer, Matthias Geiger, and Cesare Pautasso. Lessons learned from evaluating workflow management systems. In *13th International Workshop on Engineering Service-Oriented Applications and Cloud Services (WESOACS)*, Malaga, Spain, November 2017. Springer.

[164] Douglas C. Schmidt. Guest editor's introduction: Model-driven engineering. *IEEE Computer*, 39(2):25–31, 2006.

[165] E. W. Dijkstra. On the role of scientific thought. In *Selected writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982.

[166] Raman Kazhamiakin, Annapaola Marconi, Marco Pistore, and Heorhi Raik. Data-flow requirements for dynamic service composition. In *2013 IEEE 20th International Conference on Web Services, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 243–250, 2013.

[167] Christoph Evers, Romy Kniewel, Kurt Geihs, and Ludger Schmidt. The user in the loop: Enabling user participation for self-adaptive applications. *Future Generation Comp. Syst.*, 34:110–123, 2014.

[168] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[169] Luciano Baresi, Luca Mottola, and Schahram Dustdar. Building software for the internet of things. *IEEE Internet Computing*, 19(2):6–8, 2015.