Irum Rauf

Design and Validation of Stateful Composite RESTful Web Services

Turku Centre *for* Computer Science

TUCS Dissertations
No 177, June 2014

Design and Validation of Stateful Composite RESTful Web Services

Irum Rauf

To be presented, with the permission of the Department of Information Technologies at Åbo Akademi University, for public criticism in Auditorium Gamma, at ICT building, Turku, Finland, on June 16th, 2014, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3–5 A, 20520 Turku, Finland

2014

Supervisors

Professor Ivan Porres
Department of Information Technologies
Åbo Akademi University
Joukahasenkatu 3–5 A, 20520 Turku
Finland

Reviewers

Professor Gerti Kappel
Institute for Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstrabe 9-11/188-3
1040 Vienna
Austria

Professor Cesare Pautasso
Faculty of Informatics
University of Lugano (USI)
CH-6904 Lugano
Switzerland

Opponent

Professor Gerti Kappel
Institute for Software Technology and Interactive Systems
Vienna University of Technology
Favoritenstrabe 9-11/188-3
1040 Vienna
Austria

ISBN : 978-952-12-3070-7
ISSN : 1239-1883

To my family

ITHACA

When you set out on your journey to Ithaca,
pray that the road is long,
full of adventure, full of knowledge.
The Lestrygonians and the Cyclops,
the angry Poseidon - do not fear them-
You will never find such as these on your path
if your thoughts remain lofty, if a fine
emotion touches your spirit and your body.
The Lestrygonians and the Cyclops,
the fierce Poseidon you will never encounter,
if you do not carry them within your soul,
if your heart does not set them up before you.

Pray that the road is long.
That the summer mornings are many, when,
with such pleasure, with such joy
you will enter ports seen for the first time;
stop at Phoenician markets,
and purchase fine merchandise,
mother-of-pearl and coral, amber and ebony,
and sensual perfumes of all kinds,
as many sensual perfumes as you can
visit many Egyptian cities,
to learn and learn from scholars.

Always keep Ithaca in your mind.
To arrive there is your ultimate goal.
But do not hurry the voyage at all.
It is better to let it last for many years;
and to anchor at the island when you are old,
rich with all you have gained on the way,
not expecting that Ithaca will offer you riches.

Ithaca has given you the beautiful voyage.
Without her you would never have set out on the road.
She has nothing more to give you.
And if you find her poor, Ithaca has not deceived you.
Wise as you have become, with so much experience,
you must already have understood what Ithaca mean.

CONSTANTINE CAVAFY (1863-1933)
translated by Rae Dalven

Acknowledgements

When I started with my Ph.D journey, a Ph. D degree to me was what Ithaca was to Odysses. My journey to PhD degree was however not relatively so long but was nonetheless a beautiful voyage with lots of splendor and riches. The list of people that I acknowledge here is by no way complete since every person I have spent time with, every book that I have read and every travel that I have had has shaped me into what I am.

First of all, I would like to heartily thank my supervisor Prof. Ivan Porres who graciously supervised me, encouraged me and trusted me to individually pursue the challenging paths of research. Ivan is an outstanding supervisor and has been a constant source of inspiration for me to conduct quality research. His energetic and stimulating attitude towards research ideas always kept me going. I am especially thankful to him not only for being an excellent supervisor but also for being a wonderful, understanding and supportive person.

I would like to sincerely thank Prof. Gerti Kappel and Prof. Cesare Pautasso for taking the time and effort to review my thesis. Their useful comments and valuable suggestions are greatly appreciated and they really helped me in refining the final manuscript of my thesis. I am also very grateful to Prof. Gerti kappel for her kind acceptance to act as an opponent at my doctoral defence.

Furthermore, I would like to thank all my coauthors for collaborating on this research. I am grateful to Anna Ruokonen and Tarja Systä from Tampere University of Technology for their kind cooperation in research and for many useful research discussions we had. I would also like to thank my co-authors in Åbo Akademi, Ali H. Khan, Faezeh Siavashi and Dragos Truscan for sharing their domain knowledge and diverse viewpoints with me. I owe my special gratitude to Dragos for many fruitful discussions and helping me often in various matters of research and beyond.

I gratefully acknowledge the department and the Graduate School of Software Engineering (SoSE) for their financial support for my PhD and excellent research environment. The annual workshops held by SoSE in Lapland were always very constructive and relaxing at the same time. It was there that I made some good friends from research community in Finland and learnt how to enjoy research in a relaxed environment. It was a very warm experience to meet people like Kai Koskimies, Tarja Systä and Maarit Harsu from whom I learnt a lot at scientific and personal level. Thankyou for arranging such constructive environments.

I am also very grateful and honored to receive generous scholarships from Nokia Research Foundation and Ulla Tuominen Foundation.

Thanks also goes to the administrative and technical personnel at the Dept. of IT for their support in all practical things. I especially wish to thank Christel Engbolm, Britt-Marie Villstrand, Christel Donner, Nina Rytkönen, Anne-Leena Gröning and Tove Österroos for their full support in all administrative matters. Also, Joakim and Magnus for providing excellent technical support.

My stay at the Software Engineering Lab was made enjoyable due to pleasant and amusing environment in the lab regardless of work-related stress and worries. In this regard, I would like to thank all my previous and current colleagues in the lab especially Jeanette, Torbjön, Roman, Dragos, Fredrik, Faezeh, Benjamin, Marta, Adnan, Ali, Kristian, Max, Espen and Tanvir. I would especially like to acknowledge efforts of Roman in helping me with the implementation of my partial code generation tool. I am also very grateful to Fredrik and Mats Neovius for helping me with the Swedish version of the abstract.

Special thanks goes to to Prof. Zafar I. Malik, Dr. M. Zohaib Iqbal and Dr. Shaukat Ali for steering me towards the path of research, helping me with the initial steps of doing quality research and encouraging me to pursue it further.

Last but by no way the least, I would like to thank my family for their incredible support. I am especially thankful to my parents for instilling the love of education in me right from the start, for giving me a very strong foundation and on top of it supporting me against all the odds to travel abroad alone and pursue my own paths in life. Thank you for your trust and making me who I am. I am especially indebted to my mother for her countless sacrifices on all the fronts - financial, physical, emotional - to make me who I am today. I just cannot even comprehend your devotion and selfless attitude towards my growth in life. I am nothing but a fruit of your passion. I express my deep gratitude to my father, for going beyond his resources to support my education, and to both my sisters for their unwavering love and encouragement to better myself throughout. Their kind suggestions and supporting hands helped me overcome all the battles of life with ease.

My heartiest thanks goes to my dearest husband, Usman, for his unbelievable patience, absolute understanding and unwavering support. This thesis in my hand would not have been possible without your selfless love, support and encouragement to reach my goals. Thankyou for filling my everydays with laughter with your witty humor and comfort with your thoughtful gestures. You complete me!

Finally, I would like to thank my daughter Zoya, who introduced me to a totally new research area during my Phd which I totally loved. All things aside, nothing makes me more proud than calling you my daughter, already! Thanks for filling my everydays with countless blessings, for showing me the joy of finding things out with your curiosity and for reminding me to find happiness in little things of life. May you always be a reason to smile for others, as you already are, and grow up to be a source of happiness, comfort and knowledge where ever you go. Ameen.

Abstract

A web service is a software system that provides a machine-processable interface to the other machines over the network using different Internet protocols. They are being increasingly used in the industry in order to automate different tasks and offer services to a wider audience. The REST architectural style aims at producing scalable and extensible web services using technologies that play well with the existing tools and infrastructure of the web. It provides a uniform set of operation that can be used to invoke a CRUD interface (create, retrieve, update and delete) of a web service. The stateless behavior of the service interface requires that every request to a resource is independent of the previous ones facilitating scalability. Automated systems, e.g., hotel reservation systems, provide advanced scenarios for stateful services that require a certain sequence of requests that must be followed in order to fulfill the service goals. Designing and developing such services for advanced scenarios with REST constraints require rigorous approaches that are capable of creating web services that can be trusted for their behavior. Systems that can be trusted for their behavior can be termed as dependable systems. This thesis presents an integrated design, analysis and validation approach that facilitates the service developer to create dependable and stateful REST web services.

The main contribution of this thesis is that we provide a novel model-driven methodology to design behavioral REST web service interfaces and their compositions. The behavioral interfaces provide information on what methods can be invoked on a service and the pre- and post-conditions of these methods. The methodology uses Unified Modeling Language (UML), as the modeling language, which has a wide user base and has mature tools that are continuously evolving. We have used UML class diagram and UML state machine diagram with additional design constraints to provide resource and behavioral models, respectively, for designing REST web service interfaces. These service design models serve as a specification document and the information presented in them have manifold applications. The service design models also contain information about the time and domain requirements of the service that can help in requirement traceability which is an important part of our approach. Requirement traceability helps in capturing faults in the design models and other elements of software development environment by tracing back and forth the unfulfilled requirements of the service. The information about service actors is also included in the design models which

is required for authenticating the service requests by authorized actors since not all types of users have access to all the resources. In addition, following our design approach, the service developer can ensure that the designed web service interfaces will be REST compliant.

The second contribution of this thesis is consistency analysis of the behavioral REST interfaces. To overcome the inconsistency problem and design errors in our service models, we have used semantic technologies. The REST interfaces are represented in web ontology language, OWL2, that can be part of the semantic web. These interfaces are used with OWL 2 reasoners to check unsatisfiable concepts which result in implementations that fail. This work is fully automated thanks to the implemented translation tool and the existing OWL 2 reasoners.

The third contribution of this thesis is the verification and validation of REST web services. We have used model checking techniques with UPPAAL model checker for this purpose. The timed automata of UML based service design models are generated with our transformation tool that are verified for their basic characteristics like deadlock freedom, liveness, reachability and safety. The implementation of a web service is tested using a black-box testing approach. Test cases are generated from the UPPAAL timed automata and using the online testing tool, UPPAAL TRON, the service implementation is validated at runtime against its specifications. Requirement traceability is also addressed in our validation approach with which we can see what service goals are met and trace back the unfulfilled service goals to detect the faults in the design models.

A final contribution of the thesis is an implementation of behavioral REST interfaces and service monitors from the service design models. The partial code generation tool creates code skeletons of REST web services with method pre and post-conditions. The preconditions of methods constrain the user to invoke the stateful REST service under the right conditions and the post condition constraint the service developer to implement the right functionality. The details of the methods can be manually inserted by the developer as required. We do not target complete automation because we focus only on the interface aspects of the web service.

The applicability of the approach is demonstrated with a pedagogical example of a hotel room booking service and a relatively complex worked example of holiday booking service taken from the industrial context. The former example presents a simple explanation of the approach and the later worked example shows how stateful and timed web services offering complex scenarios and involving other web services can be constructed using our approach.

Sammanfattning

En webbtjänst är ett mjukvarusystem som erbjuder ett maskinlä sbart grä nssnitt till andra maskiner över ett nä tverk genom att anvä nda olika Internetprotokoll. De används alltmer inom industrin för att automatisera olika uppgifter och erbjuda tjänster till en bredare publik. REST-arkitekturen har som syfte att producera skalbara och utbyggbara webbtjänster med hjälp av teknik som samverkar bra med de befintliga verktygen och infrastrukturen för webben. Den erbjuder en enhetlig uppsättning av funktioner som kan användas för att anropa ett CRUD -gränssnitt (create , retrieve, update, and delete) hos en webbtjänst. De tillståndslösa beteendet hos ett servicegränssnitt kräver att varje förfrågan av en resurs är oberoende av en tidigare förfrågan, vilket underlättar skalbarheten. Automatiserade system, till exempel, hotellbokningssystem, erbjuder avancerade scenarier för tillståndsstyrda tjänster som kräver en viss sekvens av förfrågningar som måste följas för att uppfylla de mål som är uppsatta för en tjänst. Design och utveckling av sådana tjänster för avancerade scenarier med REST-begränsningar kräver rigorösa metoder som är kan skapa webbtjänster vars beteende man kan lita på. System vars beteende man kan lita på, kan betecknas som pålitliga system. Denna avhandling utgör en integrerad design, analys, samt valideringsstrategi som underlättar för tjä nsteutvecklare att skapa pålitliga och tillståndsstyrda REST-webbtjänster.

Avhandlingens primära bidrag är att vi erbjuder en ny modelldriven metod för att utforma tjänster för REST-webbgränssnitt och dess sammansättning. Gränssnitten ger information om vilka metoder som kan anropas hos en tjänst samt pre- och post-villkor för dessa metoder. Som modelleringsspråk använder metoden sig av Unified Modeling Language (UML) som har en bred användarbas samt välutvecklade verktyg som kontinuerligt uppdateras. Vi har använt UML klassdiagram och UML tillståndsdiagram med ytterligare designkrav för att kunna erbjuda resurs- respektive beteendemodeller för att designa REST webbgränssnittstjänster. Dessa designmodeller för tjänster fungerar som ett specifikationsdokument och den information som presenteras i dessa har mångfaldiga tillämpningar. Designmodellerna för tjänster innehåller ocksåinformation om tids- och domänkrav för tjänsten vilket kan hjälpa till med spårbarheten av specifikationskrav, som är en viktig del av vår strategi. Spårbarheten av specifikationskraven hjälper till att hitta fel i designmodellerna och andra delar av mjukvaruutvecklingsmiljön genom att spåra tillbaka ouppfyllda krav påtjänsten. Information om tjänsteaktörer ingår

också de designmodeller som krävs för autentisering av tjänsteförfrågningar av auktoriserade aktörer eftersom inte alla typer av användare har tillgång till alla resurser. Dessutom, genom att följa vårt tillvägagångssätt, kan utvecklare av tjänster försäkra sig om att de utformade gränssnitten för webbtjänster kommer att vara REST-kompatibla.

Det andra bidraget med denna avhandling är förenlighetsanalys av beteendet hos REST-gränssnitt. För att övervinna den förenlighetproblemet samt konstruktionsfel i vår tjänstemodeller, har vi använt oss av semantisk teknik. REST-gränssnitten är representerade med ett webbontologispråk, OWL2, som kan vara en del av den semantiska webben. Dessa gränssnitt används OWL 2-reasonerare för att kontrollera icke satisfierbara koncept vilka resulterar i misslyckade implementationer. Metoden är fullständigt automatiserad tack vare implementation av översättningsverktyg och de befintliga OWL 2-resonerarna.

Det tredje bidraget med denna avhandling är verifiering och validering av REST-webbtjänster. För detta ändamål har vi använt metoder för granskning av modeller med hjälp av UPPAALs modellgranskare. De tidsinställda automaterna för tjänstedesignmodeller baserade på UML genereras med vårt omvandlingsverktyg samt verifieras för deras grundläggande egenskaper såsom undvikning av dödläge, liveness (eng.), nåbarhet och säkerhet. Implementationen av en webbtjänst testas med hjälp av svart låda testning (eng., black box testing). Testfall genereras från UPPAALs tidsinställda automater och genom att använda använda online-testverktyg, UPPAAL TRON, såvalideras implementationen av tjänsten under körning mot dess specifikation. Spårbarhet av specifikationskraven behandlas också vårt valideringstillvägagångssätt med vilket vi kan se vilka servicemål som är uppfyllda och spåra tillbaka de uppfyllda servicemålen för att upptäcka fel i designmodellerna.

Avhandlingen sista bidrag är en implementation av ett beteende REST-gränssnitt och tjänstövervakare från designmodeller för en tjänst. Det ofullständiga kodgenereringsverktyget skapar en kodstomme för REST-webbtjänster med pre- och postvillkor för metoder. Pre-villkoren för metoder tvingar användaren att åberopa tillståndsstyrda REST-tjänster under rätta förutsättningar och post-villkoren tvingar tjänsteutvecklare att implementera rätt funktionalitet. Detaljerna i metoder kan vid behov manuellt fyllas i av utvecklaren. Vi strävar inte efter full automation eftersom vi bara fokusera pågränssnittsaspekter hos webbtjänsten.

Användbarheten av metoden demonstreras med ett pedagogiskt exempel påen tjänst för bokning av hotellrum och ett relativt komplext exempel påen tjänst för semesterbokning tagna från industrin. Det förstnämnda exemplet visar en enkel förklaring av metoden medan det senare exemplet visar hur tillståndsstyrda och tidsbestämda webbtjänster som erbjuder komplexa scenarier och involverar andra webbtjänster kan konstrueras med hjälp av vår strategi.

List of original publications

1. Irum Rauf, Faezeh Siavashi, Dragos Truscan and Ivan Porres, *An Integrated Approach to Design and Validate REST Web Service Compositions*, In Proceedings of WEBIST 2014, 10th International Conference on Web Information Systems and Technologies, pages 104-115, SCITEPRESS Digital Library, 2014.
2. Ali Hanzala Khan, Irum Rauf, Ivan Porres, *Consistency of UML Class and Statechart Diagrams with State Invariants*, In Proceedings of MODELSWARD 2013, 1st International Conference on Model-Driven Engineering and Software Development, 1, pages 1-11, SciTePress Digital Library, 2013.
3. Irum Rauf, Ali Hanzala Khan and Ivan Porres, *Analyzing Consistency of Behavioral REST Web Service Interfaces*, In Proceedings of EPTCS 2012, 8th International Workshop on Automated Specification and Verification of Web Systems, Electronic Proceedings in Theoretical Computer Science, pages 1-15, 2012
4. Irum Rauf and Ivan Porres, *Towards Behaviorally Enriched Semantic RESTful Interfaces using OWL2*, In Proceedings of ICWE 2011, the 11th International Conference on Web engineering, pages 407-410. Springer-Verlag, 2011.
5. Irum Rauf and Ivan Porres, *Beyond CRUD*, REST: From Research to Practice, pages 117-135, Springer, 2011.
6. Irum Rauf and Ivan Porres. *Designing Level 3 Behavioral RESTful Web Service Interfaces.* ACM SIGAPP Applied Computing Review 11, no. 3, pages 19-31, 2011
7. Ivan Porres and Irum Rauf, *Modeling behavioral RESTful web service interfaces in UML*, In SAC 2011, Proceedings of 26th Annual ACM Symposium on Applied Computing Track on Service Oriented Architectures and Programming, pages 1598-1605, 2011.
8. Irum Rauf, Anna Ruokonen, Tarja Systa, and Ivan Porres, *Modeling a Composite RESTful Web Service with UML* In ECSA 2010, Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, pages 253-260, ACM, 2010.
9. Ivan Porres and Irum Rauf, *From Nondeterministic UML Protocol Statema-*

chines to Class Contracts, In Proceedings of ICST 2010, Third International Conference on Software Testing, Verification and Validation, pages 107-116 , IEEE Computer Society Washington, DC, USA , 2010.

10. Ivan Porres and Irum Rauf, *Generating Class Contracts from Deterministic UML Protocol Statemachines*, In Models in Software Engineering, pages 172-185, Springer, 2010.

Contents

Chapter 1

Introduction

Web services are autonomous piece of software that have a machine processable interface and provide their functionality over the network. The users of web services are other web services or interactive applications like web browsers or mobile applications that can use this information to do other tasks without much human intervention. This is in contrast to the behavior of the document web or interactive web applications that are designed for human users who search for a particular information, receive it in a format that they understand and then use it manually as input to another software or machine for further processing. Over the past decade, development of web services and their compositions has gained much attention in the software industry and academia and it is evident by the large number of web services available over the web [9]. More and more companies are using web services to expose their software functionality to a wider audience and to automate their existing tasks.

The Representational State Transfer (REST) architectural style has been introduced by Roy Fielding in 2000 [52] and has become a popular approach to design web services. REST outlines the architectural principles to build Internet-scale distributed hypermedia systems. This has encouraged a number of users on the web and big enterprises to use REST web services. Although REST web services advocate to be simpler to implement [52], when compared with SOAP-based web services, their use in advanced and complex scenarios may require careful design and validation practices for developing REST web services that can be trusted for their functionality. Such web services impose certain restrictions on how the service should be created and used. These restrictions should be considered during the development as there may not be necessarily a human developer on the other end to process and figure out what a service does via trial and error method. Thus, REST web services need to be designed carefully for such scenarios, keeping in mind different constraints it imposes.

In this thesis, we have given an integrated approach to design, analyze and validate web services and their compositions that comply to REST architectural

style and are implemented for advanced scenarios. We aim to facilitate the development of verifiable RESTful web services through modeling and model-driven engineering techniques.

In this chapter, we present an overview of this thesis. First, we present the types of web services on the basis of their architectural styles in section 1.1. This is followed by details of resource oriented architecture and properties of REST web services in sections 1.2 and 1.3, respectively. The notion of states of a service and statelessness of a protocol is explained in section 1.4. The motivation behind our work is presented in section 1.5. We then provide an overview of our work along with the research questions and contributions in section 1.6. In section 1.7, we present a brief overview of the technologies on which our work is built upon. The adopted research methodology is explained in section 1.8. The chapter is concluded in section 1.9.

1.1 Types of Web Services

Web services can be classified based on the design principles used to develop them, i.e., the architectural style they are built upon. Web services are usually classified based on two main architectural styles: SOAP-based and REST-based. We call SOAP based web services *Big Web Services*, following the naming convention first introduced in [111] and REST-based services as REST web services.

1.1.1 Big Web Services

Big web services are based on WS-* protocol stack (SOAP, WSDL, etc.) and are operation centric. The service exposes its functionality in the methods that can be invoked on it. The user of the service understands these methods via service description or via the descriptive names of the methods that are offered by the service interface. The interpretation of these messages is left to the service that receives them. The SOAP messaging protocol that is used to transfer the messages does not impose any application semantics on them. This means that the semantics of applications are maintained within the boundaries of the service and are determined by the message payloads (header and body content) [127].

1.1.2 REST Web Services

REST web services are built on the principles of REST architectural style [52]. REST architectural style outlines the principles and constraints of web architecture that builds Internet-scale distributed hypermedia systems.

REST advocates stateless interaction between components, i.e., every request is independent of its previous request with no stored context on the server. This allows REST web services to cater large number of clients resulting in system scalability since the provision of not having to store state between request allows the server

to free resources rather quickly. This may affect the system as a design trade-off resulting in decreased network performance due to data repetitions. However, REST web services play well with existing infrastructure of the web, e.g., caching, clustering and load balancing that can help in improving efficiency of the network.

REST is centralized around the concept of resources which are pieces of information that can be navigated through URIs. The main features offered by REST include identifying resources with names, manipulating resources with a uniform interface, using hypermedia to link these resources and using stateless interaction between client and server. With the help of these features, REST web services can serve a large number of users and integrate well with other technologies of the web.

1.2 Resource-Oriented Architecture

Resource-Oriented Architecture (ROA) [111] is a structural design that fulfills design criteria presented by REST architectural style. It aims to clear the ambiguities in REST design principles by presenting a structural design that applies these principles. ROA is based on the following REST concepts: resources, their names, their representations, and links between them. Below we give a brief introduction to these concepts and method semantics in REST.

Resources: A REST web service exposes its functionality as a set of resources. A resource is any piece of information that can be the target of an interaction and is defined by Fielding [52] as an *intended conceptual target of a hypertext reference*. Every resource must have at least one URI [90], where URI gives the name and address of the resource.

Representations and HATEOAS: When a URI is invoked on a resource, it returns a representation of the resource that defines its state. This representation is in the form of an XML or JSON file that contains information about its attributes and the links that can be taken further. These links connect resources and communication can move forward by exchanging the states of the resources. This establishes the notion of *hypermedia as the engine of application states* (HATEOAS) commonly used to define REST architectural style. The service moves forward through different states during its lifecycle by exchanging states of the resources as hypermedia links in resource representations.

Method Semantics: Clients interact with resources over the web. The verbs that interact with these resources and manipulate the information provided by these resources and their representations are given by methods. REST architectural style requires that the same set of methods should be called on different resources. HTTP is a protocol that forms the basis of web and implements well the principles

of REST [52]. Though, technically, it is one of the interaction protocols that can be used to interact with resources over the web but due to its pervasiveness it is considered to be *the* protocol of the web [111].

CRUD (create, retrieve, update and delete) operations can be performed on resources using standard HTTP methods. These HTTP methods are considered as application-level constructs that the programs can use to interact with another program over the network in a standard manner with well-defined semantics [127]. The HTTP request is targeted to a resource via a URI of that resource and is returned with an HTTP response. HTTP response consists of HTTP response code, response headers and representation of the resources. Response headers provide the operating parameters and representation of the resource is the document that gives information about the resource. The HTTP response code is a numeric code that tells the clients whether the request went successful or not. HTTP has a list of status codes that reveal how the request went [27], for example, 200 means the request was successful, 404 means the resource was not found and 403 implies that it is forbidden to make this request on this resource. The client machine interpret these response codes to know how their request went.

1.3 Properties of REST Web Services

REST web services exhibit the following four properties [111]:

1.3.1 Addressability

A REST web services exposes the information it considers servable to its clients as resources. These resources can be reached only via a URI else a client has no information about its existence. The addressability feature requires that every resource should have atleast one URI.

1.3.2 Connectedness

This feature implies that the resource representation not only contain data about resource attributes but can also contain links to other resources. These links connect resources to each other and service client gets an experience of connectivity between resources, i.e., moving from one resource to another.

1.3.3 Uniform Interface

It requires that there should be a same set of methods, with predefined semantics, that can be invoked on all resources. In REST web services all resources are manipulated using the standard HTTP methods. The HTTP GET, POST, PUT and DELETE are used to retrieve information from a resource or change its state.

1.3.4 Statelessness

Every request from the client should contain all the information that is required to process it, i.e. the server is not responsible of keeping any context information with it. Hence, every request is treated independently.

With these features, REST web services can play well with the existing tools and infrastructure of the web. The feature of connectivity and uniform interface allows use of existing tools and infrastructure like web crawlers, curl, caches etc. The addressability requirement helps us to create extensible web services. The extensibility feature enables to add a functionality to the system without impacting the rest of the system [52]. URI addresses can be constructed in an hierarchical manner, such that they make data structure and relationships. Thus, it becomes convenient to add-in more functionality into the existing structure without modifying it. The statelessness requirement simplify the development of systems that can handle many service requests simultaneously facilitating scalability since the server does not need to keep any context information and the service requests can be handled by different servers. Currently, REST web services are widely adopted in the web and have numerous users, including enterprizes such as Google, Yahoo, Amazon and Flickr.

1.4 Stateful Services vs Stateless Protocol

Web services can have different service states that a service must go through during its lifecycle. A stateful service requires a certain sequence of method invocations that must be followed in order to fulfill the functionality a service promises to deliver to its users. For example, in a room booking service, the booking cannot be paid until a booking is made. This requires that a booking must be made first and then it should be paid. If the user of the service does not follow this protocol, it cannot expect the desired results. In a stateful service, the result of a (side-effect) method is dependent on the current state of the web service or resource (in case of REST web service). A method invoked on a service or a resource , may return different results depending on the state of the service or resource. For example, consider the case of a service that allows only canceled bookings to be deleted. In such a case, the result of invoking a method that deletes a booking, on a booking instance (or resource) that is canceled, is different from the results of invoking the same method on a booking instance (or resource) that is not canceled. In the first case the booking is deleted but in later case the booking is not deleted as it is still an active booking. A state of the service is thus defined as a specific condition of the service at a certain time instance.

Web services follow a typical client server architecture that provides a platform-independent and language-independent mechanism to transmit messages over a network. A server receives a request from the client. This request could be a single request followed by a response. For a stateful service, this could be a

series of message requests. This message exchange happens via a communication protocol that can be either stateful or stateless. A stateful protocol requires that the server can associate a request with the previous requests and knows that they all come from the same user. On the other hand, a stateless protocol treats each request independently and unrelated to the previous request. Figure 1.1 represents graphically the difference of communication between a stateful (left) and a stateless (right) protocol for opening and reading a file. A stateful protocol requires to keep a connection of the opened file along with the information on the last position of the cursor, on the other hand a stateless protocol has all the information in the method parameters and does not need to maintain any state information from the previous request. This of course comes with the overhead of opening and closing the file again and again but offers a scalable architecture since the service does not need to keep any context information and service requests can be handled simultaneously by different servers.

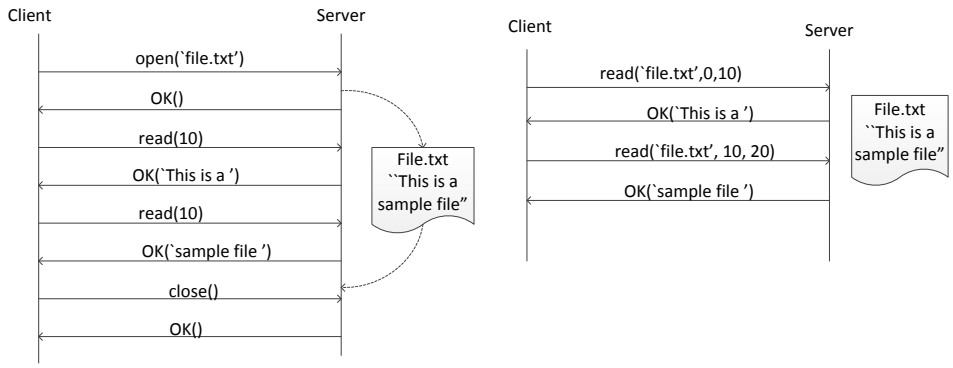


Figure 1.1: (left) Stateful Protocol (right) Stateless Protocol

1.4.1 Stateful Services as "Big Web Services"

Big web services use different specifications built on top of each other to address different tasks. Since there is no notion of states in web services, WS-Resource Framework [10] and WS-Transfer [48] are commonly used to model state in *big* web services [55]. They are both almost similar technically. The architecture style they use to store state information consists of storing state of the service as an XML document and give it an address via a WS-Addressing EPR (End Point Reference). EPR defines the address for a resource in a SOAP header and is defined in WS-Addressing specification [30].

The main drawback of using a such an approach is that while processing a request, the server needs to retrieve any kind of service context or state. Also, such an approach can make things complicated since the service requires a lot of upfront consideration to efficiently store and enable session information to maintain states.

Stateful services may also facilitate transactions. Transactions have their own set of well-defined properties and require that a certain sequence of operations be treated as a unit of work that is either completed fully or canceled altogether. In order to support transactions *big* web services use specifications like WS-Transactions [23] and WS-Coordination [37].

1.4.2 Stateful Services as "REST Web Service"

The REST web service uses HTTP as a stateless protocol for communication between the server and the client. This constraint leads to the construction of scalable web services since the server does not need to preserve any session or state information in between the client requests. All the data needed to fulfill the service request is part of the request so the intermediary servers may forward, route and load balance without requiring server to hold any state in between the requests with an aim to decrease the overall response time of a web service call.

Creating stateful services using a stateless protocol is an interesting design challenge since there is no provision of passing or maintaining hidden session information over a sequence of events. Some authors claim that REST supports totally stateless operations and if an operation needs to be continued, then REST is not the best approach and SOAP may fit it better [53]. Some authors propose keeping a *transient state* (*soft state*) to temporary keep the state of the service somewhere that is destroyed or updated once that state is traversed and no longer useful [132]. Another option could be storing the state of the service in cookies. Cookies are small pieces of data that can be stored on user's browser to take the load off the server of saving user specific information. However, use of cookies may not be an optimal solution to save service state since they can misrepresent user information and can be a security and privacy risk [52].

However, inspite of all these discussions, REST web services come with the property of transfering state of the application (service in our case) from one resource to another. REST does that by providing links in the representation of the resources [127]. These links contain information on what further links should be addressed so that the sequence of method invocations is maintained and also the state of the service is preserved. Thus using a stateless HTTP protocol, services that give stateful behavior can be constructed in this manner. The objective behind this is to create stateful scalable REST web services.

The support for transactional interactions with REST web services has also started gaining interest in the research community. Marinos et al. [88] provides a transaction model that satisfies the contraints of the transactions and also of REST architectural style. Razavi et al. [110] uses isolation theorems to propose an approach for RESTful transactions. Kochman et al. [74] describe a system architecture and algorithms for batched transactions for REST web services. To support distributed atomic transactions over REST services, Pardon and Pautasso [98] present a light-weight protocol, based on Try-Cancel/Confirm (TCC) pattern

which assumes that reserved resources are either confirmed or canceled within a given time.

1.5 Motivation

The features offered by REST web services simplify the overall architecture of the system offering many non-functional properties. However, this also opens new research questions when REST web services are used in advanced scenarios that require more than just simply retrieving and manipulating information from the database.

1.5.1 Service States

Stateful services require the service users to follow a protocol as a set of sequence of events that should be followed in order to fulfill service functionality. These services can be called stateful services as explained in Section 1.4. All REST web services are stateful by nature since CRUD (create, retrieve, update and delete) methods can be called on every resource to change or retrieve its state. However, from the developer's perspective, when REST web services are used in advanced scenarios, it may become a challenge to design them in a consistent and verifiable manner, since more advanced the scenarios, the more careful design efforts are needed to communicate the right information to the right users.

In RESTful web services, the state of resource determines the result of an invocation and the resource representation contains information on what further links (representing state transfer) can be followed by the service user. However, when designing such services, developers need to carefully design what links can be part of the resource representation, since a resource can give a different representation as response to a method invocation depending on the current situation of the service. For example, the response of invoking a PUT on *cancel* resource for an unpaid booking is different from the response received by invoking PUT on *cancel* resource of a paid booking. The former will cancel the booking and give links to rebook the room, and in the latter case, the response would provide links to get the payment refunded or autonomously initiate a payment refund service giving links to either browse elsewhere while waiting for the payment confirmation or give a confirmation response (depending on the design of the service). In both the cases, and other similar scenarios, the resource representations need to be carefully designed so that they transfer the right state of the service, i.e., service state. We define service state as a predicate over resources. This work complements the work done on transactional services. However, we are not using the word transactional services as the focus of our study is not to specifically address all the transaction principles, i.e., Atomicity, Consistency, Isolation and Durability, but to facilitate the designing and development of REST services for advanced stateful scenarios, from the developer's perspective. Our work on designing stateful services should

be considered complementary to transactional services. Similarly, the usage of Try-Cancel/ Confirm protocol [98] for transactional REST web services can be used in conjunction with our approach and vice versa.

1.5.2 REST Web Service Composition

Web service compositions may also offer complex scenarios. A web service composition is a process in which new web services are composed with specific business goals from existing web services that are already published over the Internet. The functionality of newly composed web services is dependent upon the functionality of existing web services. Composite web services have their own set of unique requirements that must be fulfilled in order to fulfill the functionality it advertises, such as:

- *Timing Constraints:* Composite web services may impose timing constraints on partner web services as in the case of transactional compositions that assume that the sequence of events, that make changes to information that a web service holds, either succeed as a complete unit or fail. This information is important to be taken into account when designing and developing such web services.
- *Service Actors:* Web service composition may also involve different actors, machines or human, who can invoke methods on different web services. Not every actor may be allowed to invoke every method on every service. A web service developer needs this information in order to implement web services that do what they are required to do and not do what they should not do.

We are interested in creating web service compositions that provide REST interface features alongwith their own set of requirements.

1.5.3 Comprehensible Information

In addition, the more advanced or complex the scenario would be, the greater number of resources and the relationships between them will arise. In such a case, keeping a track of how resources are connected, how they contribute towards services states and what are the allowed and not allowed methods on different resources can be difficult to manage. Thus, keeping a track of these relationships and understanding them in a way that is comprehensible and communicable becomes important.

1.5.4 Behavioral REST Interface

Another aspect of web services that motivates our approach is the provision of a behavioral REST web service interface. Web service interfaces provide methods that can be invoked on it. However, as in the case of stateful services, a web service may constrain its users to invoke these methods in a certain order to obtain

functionality that is expected from it. This information is usually not present in a service interface that only provides information about the methods that can be invoked on it along with details of how to use it in text format in some cases. We are interested in providing a behavioral interface for REST web services that can constrain the service user to invoke the service under right condition and also constraints the service developer to implement the right functionality.

All these constraints and requirements open new research dimensions when combined with REST constraints on web services. For the service developers, designing and implementing such services for advanced scenarios, that deliver the advertised functionality, can become an uphill task since there can be lots of information that needs to be handled in a meaningful way.

1.6 Overview of the Approach and Contributions

Developing REST web services that are dependable in the sense that they can be trusted with greater confidence in their functionality is the goal of this thesis. We are interested in exploring this research area and give an integrated approach that facilitates the service developer to design, analyze and validate REST web services and their compositions that require the service user to follow a protocol. The created web services should be dependable such that they can be trusted with higher confidence in their advertised functionality.

Our research thesis is based on four main research areas: *Design, Consistency Analysis, Validation and Implementation*. We address different research questions in each of these areas. Our study of the current literature show that many efforts have been made for the development, design and validation of REST web services. These works are detailed in different chapters of this thesis. However, we feel the need for the development approaches that can facilitate the task of developing REST services for advanced scenarios and in doing so we have contributed not only in the overall domain of REST web services but also individually in different areas of design, verification and validation of web services, as mentioned in their individual chapters. The created services should be dependable in the sense that the service developer can design such services in a manner that they can be trusted to provide RESTful behavior and the advertised functionality.

Below, we present the different research questions posed in each area, our contribution and an overview of how and in which chapter/ chapters of this thesis we elaborate them in detail. An overview of our integrated design and validation approach for stateful REST web services is given in Figure 1.2. In Table 1.1 we show to which area our research papers belong to and they answer which research questions.

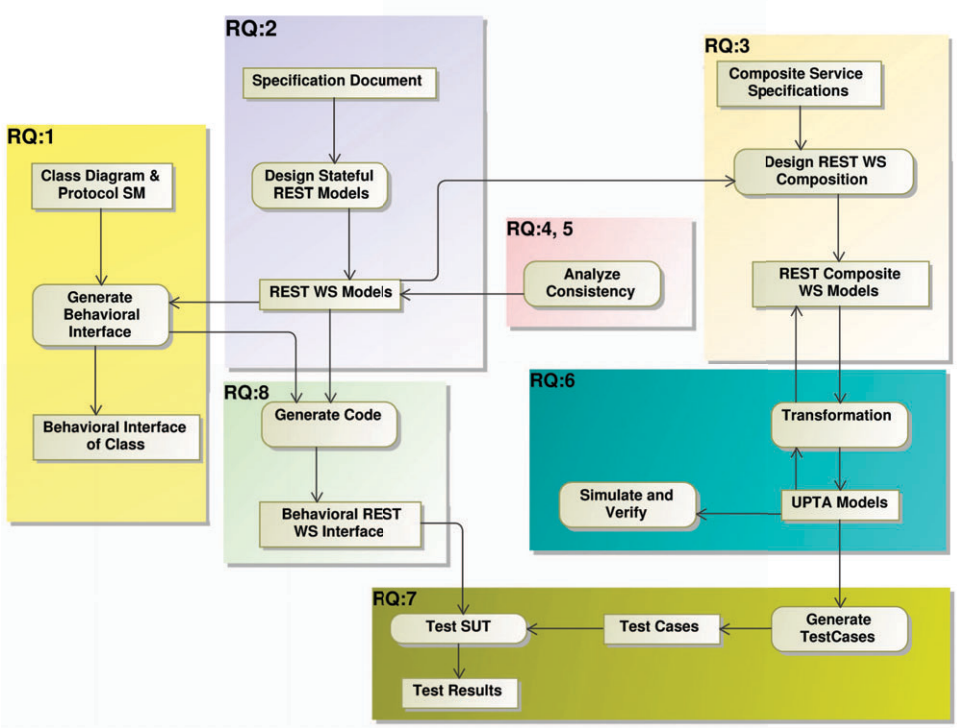


Figure 1.2: An Approach to Design, Analyze and Validate Stateful REST Web Services and their Compositions

1.6.1 Design

To start with, we are interested in providing a design approach using models that include all the information required to build stateful REST web services, as motivated above. The models provide a graphical representation of the specifications of the system under development that can be comprehended and communicated with relative ease among different stakeholders. They can provide representation of service specification from different perspectives that can lead to better understanding of the system. Our aim is to use UML (Unified Modeling Language) which is well accepted in the industry and academia and has many well-known and mature tools with a wide user base. Also, it can target design requirements independently of the implementation details. In our approach, a service can invoke other services and exhibit stateful and timed behavior while still complying with the REST architectural style.

RQ 1: How to describe and automate the generation of a behavioral interface?

RQ 2: How to design behavioral interface specifications of REST web services with stateful behavior?

RQ 3: How to design composite REST web services?

Contribution: We design behavioral interfaces for web services with advanced and complex scenarios that are REST compliant using UML. These behavioral interfaces are modeled with UML class diagrams as resource models and UML state machine diagrams as behavioral models. These models have direct mapping to the machine-processable REST interfaces. We have also modeled information about different service actors who are authorized to access different resources. This information facilitates the authentication mechanism of the service. In addition, service goals are labeled as service requirements on the behavioral model specifying when, during service lifecycle, a certain service requirement is fulfilled. The time constraints imposed on services are also modeled with time events in UML state machine. We have also extended our design approach to support the composition process of the REST web services and provided behavioral interface specifications for a REST web services.

Elaboration: The design approach for creating behavioral REST interfaces is detailed in Chapter 2 to 5. The design models for composite REST web services are presented in Chapter 7.

1.6.2 Consistency Analysis

The service design models represent the system from different perspectives and due to human error they may contain contradicting specifications of different models of the same system or they may have specifications that cannot be satisfied in any implementation. Thus, the service models should be analyzed for their consistency to ensure that the designed models do not have unintended errors. We aim to analyze the consistency of service design models using semantic concepts and OWL 2 reasoners. We address the following research questions in this area:

RQ 4: How to represent service design models as a web ontology?

RQ 5: How to analyze the service design models of REST web services for their consistency?

Contribution: We represent the structure of resource and behavioral models in ontology language OWL2 and provide tool support for UML to OWL2 translation. The OWL 2 ontology of service design models is passed to an OWL2 reasoner that provides report of unsatisfiable and satisfiable concepts. Unsatisfiable concepts will reveal the *resource definitions* that cannot be instantiated or behavioral states that cannot be entered. The reasoning of OWL 2 ontologies is supported by the the OWL 2 reasoning tools already available in the industry.

Elaboration: Chapter 6 presents our work on consistency analysis of service design models.

1.6.3 Validation

Validating service design models and service implementations for their correct behavior builds confidence of the developer that the services are designed correctly and the implementation is delivering the right functionality. Since we are interested in creating web services that can be trusted to provide correct functionality, so validation becomes an important part of our approach. The design models should be verified for their basic properties like deadlock freedom, liveness, reachability and safety. The service implementation should also be tested for its functional and temporal properties.

RQ 6: How to verify that the service design models of stateful and timed REST web services are built correctly?

RQ 7: How to validate the implementation of REST web services against their specifications?

Contribution: In order to validate the dynamic and timed behavior of the service design models, we have used the model checking approach. Models are translated into UPPAAL timed automata (UPTA) in order to make them comprehensible for UPPAAL model checker [81]. UPTA are verified with UPPAAL for their basic properties like deadlock freedom, reachability, liveness and safety. Performing the verification of the web service composition in a model-checking tool allows us to increase the quality of the specifications before proceeding to the implementation. We have also validated the implementation of a REST web service against its specifications using UPPAAL TRON tool [81] which is a black-box conformance testing tool for the timed systems. In addition, our validation approach also provides requirement traceability by checking which of the service design goals are met and which are missed by the service implementation.

Elaboration: The verification and validation mechanism of a composite REST web service is presented in Chapter 8.

1.6.4 Implementation

Model driven engineering [99] advocates generation of code from the models to reduce time and efforts needed during the development phase. An automated process that can create behavioral interface skeletons of REST web services can facilitate the service developer in the creation of REST web services in an auto-

Table 1.1: Research Questions and Publications

Research Area	Research Question	Publication
Design	RQ: 1	9, 10
	RQ: 2	6, 7
	RQ: 3	1, 8
Consistency Analysis	RQ: 4	4
	RQ: 5	2, 3
Verification & Validation	RQ: 6	1
	RQ: 7	1
Implementation	RQ: 8	5

mated manner. We require to generate code skeletons for the behavioral REST web service interfaces directly from the models.

RQ 8: How to generate code skeletons for a behavioral REST web service interface from the design models?

Contribution: The partial code generation tool is implemented in Django web framework [66]. It generates code skeletons with the pre- and postconditions for the service methods. The tool takes service design models as input.

Elaboration: The details of our service implementation and partial code generation tool are presented in Chapter 9.

In the next section, we give background of approaches and technologies we build our work upon.

1.7 Background

1.7.1 Unified Modeling Language (UML)

A model represents a system under development in a simplified manner focusing more on relevant design decisions and ignoring the unimportant details that are not considered part of the problem. UML has emerged as a standard modeling notation that provides model representation of the system in an abstract manner from different perspectives [125]. The importance of a standard modeling notation cannot be ignored since it provides many benefits to the system developers. The communication between different development teams is simplified since a common language can be used for communication and also due to a large user base mature and sophisticated tools are available that constantly improve with time. In addition, these models can serve as a part of the specification document.

The UML standard provides different types of diagrams that can be used to document a software system such as class, state, sequence and deployment

diagrams [125]. These diagrams model a system from different viewpoints. For instance, sequence diagrams model object interactions and class diagrams represent the static structure of a system.

1.7.2 Web Ontology Language

Semantic web aims to enable machines to process, combine and infer information in a meaningful way from the data. Technologies like ontologies, inference etc. attempt to standardize this information sharing mechanism so that they can be more easily supported by a software. Ontologies provide representation of a set of concepts in a domain, their properties and the relationships between those concepts. Web Ontology Language 2 (OWL 2) [93] is one of the languages commonly used to define ontologies.

OWL2 provides mechanism to define classes, properties, relationships, constraints and axioms that are stored as semantic web documents defining a particular domain of interest [93]. OWL2 has its formal underpinning in description logic which makes it possible for applications to reason over the facts expressed as axioms in the ontology. There are several reasoning tools available for OWL 2 like Pellet [118], Hermit[108], etc. These reasoning tools can generate new information by processing facts captured in OWL 2 ontology [127].

1.7.3 Model Checking

Model checking is a way to exhaustively and automatically check if a finite-state model of a program satisfies its specifications [46]. The goal is to see whether the models contain safety requirements like deadlock and other critical properties that can cause a system to crash.

UPPAAL is a commonly used model checking tool for verifying real time systems through modeling and simulation [82]. It is designed based on timed automata and includes other features like integer variables, structured data types, user defined functions, and channel synchronization [16]. A real-time system can be modeled by one or several timed automata that work in parallel. Each automaton is composed of nodes (location), edges (transition), clocks and variables representing different properties of the system. At a time, the system is in one state, which is defined by all the current locations of the automata, all variable values and the clock values. System updates the state by executing a transition from the current location to another location. The transition can be fired separately or parallel with another automaton.

A channel is a synchronization feature in UPPAAL. Two edges in different automata can synchronize if one is emitting and the other is receiving on the same channel. Synchronization between automata can also be provided by a clock. The clock is a type of variable with non-negative real numbers and it can be defined as a local variable in each automaton transition or as a global variable. The global

clock can be updated by all automata in the model, while the local clock can be updated only by the corresponding automaton.

1.8 Research Methodology

A research methodology defines a systematic approach undertaken in conducting a research. Design science is one such research methodology that answers questions like whether it is possible to build a certain innovation and how useful that innovation can be [73]. Our work provides a novel approach to build web services for complex and advanced scenarios that offer RESTful features. We use UML models at the design phase and compliment them with validation approaches and different tools to provide a model-driven engineering solution for the problems faced by the service developers in creation of RESTful services for advanced scenarios. In this context, the research presented in this thesis follows a design science approach.

Figure 1.3 shows the design science philosophy with a sequential process for an artifact (i.e., a construct, a model method or an instantiation) [73] given by March and Smith [87]. In the first phase, an artifact is built to perform a specific task. We, then, evaluate the artifact by using it to see if it works. The final stage of demolishing the artifact means that either the use of the artifact is finished or there is a transition from the use of an old artifact to the new one.

In our research thesis, we give a new design strategy to build models that are RESTful and have not been designed with such characteristics before in the literature. This maps to the first stage of Figure 1.3. Thus, according to our knowledge and literature survey, our work presents a novel engineering solution to the research problems we have highlighted earlier.

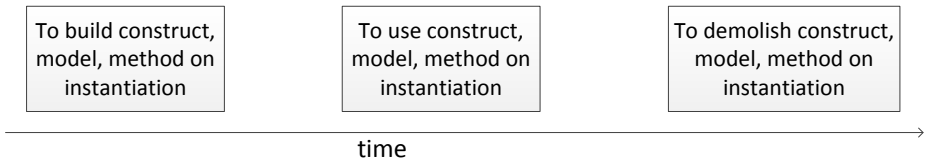


Figure 1.3: Sequential processes for Design Science [73]

The design and validation approaches were used on a relatively complex worked example to see the applicability of the approach and see if it works for real life problems. This maps to the second stage of the design science approach according to the Figure 1.3. The last stage of the design science research methodology talks about demolishing of the artifact. Demolishing may mean that either there is a transition from the use of an old artifact to the use of a new one or it may mean that the use of an old artifact is finished [73]. In software development paradigm, sometimes software artifacts can also be reused [56]. Similarly, in our case, either the use of the designed artifacts will be finished, replaced by new ones

as the design evolves or they can become part of the specification document and saved in the repository for future reuse.

1.9 Conclusion

In this chapter, we present an overview of our thesis and present the concepts on which our thesis is based upon. We briefly explained different types of web services based on their architectural style and presented the conceptual underpinnings of stateful services and stateless protocol along with their differences. The technological background of our work is also presented in this chapter, that includes UML, OWL 2 and the model checking paradigm. The motivation behind our work is to provide service developers with a holistic approach that spans multiple phases of the development cycle in order to create dependable REST web services. The research questions formulated based on these research goals are also presented along with the research methodology that was adopted to conduct this research.

Chapter 2

Designing RESTful Web Services in UML

A web service interface should have the properties of any software interface, i.e., it should define the services offered by a software system without revealing its implementation details. In the context of a RESTful web service, the interface should include information about the available resources, their addresses and their representations. It should show how the resources are connected, what methods are supported in each resource and what is the outcome of invoking a method in a specific resource. These and other requirements (motivated earlier) for creating behavioral interface of REST web service need specific design decisions.

In this chapter we give a detailed overview of our design strategy in order to create REST compliant web services. In the Section 2,1, we present the design requirements that should be addressed in the behavioral interface specifications of a REST web service. This is followed by a brief overview of our design approach, in Section 2.2. Applications of behavioral interfaces are discussed in Section 2.3 and the related work is presented in section 2.4. We conclude our chapter in section 2.5

2.1 Requirements for behavioral REST web service interface

We require that the REST interfaces that we design should exhibit REST features and should be fully compliant with the REST hypermedia principle. The designed models should also provide behavioral information about interface methods and their usage. In addition, the designed interfaces should provide information about the users who are authorized to invoke service methods, the service goals and should also address timed behavior of web services.

2.1.1 REST Interfaces

The REST architectural style is defined by four attributes. In the context of a web service these attributes are:

- *Addressability*: The REST style requires that any important piece of information related to a service should be exposed as a resource and each resource should be addressable via a URI.
- *Connectedness*: This requires that the resource representation contains links to other resources.
- *Uniform Interface*: All resources are manipulated using the standard HTTP methods. The HTTP GET, POST, PUT and DELETE are used to retrieve information from a resource or change its state.
- *Statelessness*: There is no hidden session or state information. Besides, the effects of the POST, PUT and DELETE operations should be observable in the affected resources.

Any RESTful web service should comply with these four attributes. We impose these attributes as requirements over the design of our web service interface.

2.1.2 Richardson Maturity Model

HTTP is a stateless protocol and each HTTP request is treated independent of any previous request. However, interactive web services often require that the state of the service is preserved such that a new request is in relation to the previous request. Such services take the client through a sequence of HTTP requests in a particular order to fulfill a task. RESTful web services take HTTP as an application protocol and requires that the same set of methods (HTTP verbs) is invoked on each resource offering uniform interface. However, in order to create RESTful web services with stateful behavior, there should be a provision to carry the state of the service from one independent HTTP request to another. The best way to do this is to provide links in the representations of resources [127]. These links can contain information from the server on what further links should be addressed so that the sequence of method invocation is maintained and also state of the service is preserved. Thus, using stateless HTTP protocol, stateful services can be constructed in this manner. The objective behind this is to create web services that are REST compliant and offer advanced scenarios.

However, not all web services are created in a manner that fully employ the potential of web, not even many services that claim to be RESTful. In order to identify maturity of REST web services, Richardson Maturity Model (RMM) [127] presents a classification of web services to quantify the maturity of web services.

- Level 0 services are the basic level services that use a single URI and a single HTTP method.
- Level 1 services have many addressable resources but use only single HTTP verb for all the resources.

- Level 2 services use several URI addressable resources and support several HTTP verbs on the exposed resources.
- Level 3 services, in addition to the URI addressable resources and support of several HTTP verbs, contain URI links to other resources in their representations that might be of interest to the consumer of the service.

In this classification of web services, the level 3 services are said to be most web-aware since they take hypermedia as the engine of application states. We require that web services created using our design approach are fully compliant with REST having maturity of level 3 according to RMM.

2.1.3 Method Contracts

The interface of a web service advertises the operations that can be invoked on it. A web service developer looking for a particular service finds the service over the web and integrates it with other services by invoking the advertised operations and providing it the required parameters. These operations may require a certain order of invocation or there may be special conditions under which they can be invoked. These conditions, i.e., pre- and post-conditions of a method are called contracts. This information, together with the expected effect of an operation forms part of the behavioral interface of a service. The role of contracts as behavioral interfaces has been investigated for software classes [91, 45, 35] and also in the domain of web services [63, 47].

We require a design approach that preserves the sequence of method invocations and contains behavioral information specifying the conditions under which the methods can be invoked and their expected results. This information can be extracted from the models and asserted in the interface description language. These assertions constraint the service user to invoke the methods under right conditions and also constraint the service implementation to provide the functionality that is expected from it.

The behavioral interface can also be used to test a service implementation and for service discovery. More advanced scenarios, such as automatic service discovery and service repositories rely on formal descriptions of services.

2.1.4 Authorization

The web services need to secure information in order to provide its users data integrity and confidentiality. A secure web service depends on many different techniques and technologies that work together to provide security. In the case of *big* web services, different higher-order protocols address issues like identity and trust. *Identity* refers to the ability of the system to authenticate the parties involved in the transaction and *trust* concerns authorizing the party to interact with the system in a prescribed way [127]. REST web services use HTTP protocol and can provide authentication mechanism using HTTP basic authentication, HTTPS,

or HTTP Digest authentication. The stateful REST web service may involve different parties that may or may not be authorized to invoke specific resources. The users need to be authenticated to access different resources. This authorization information can be sent in the authorization header of the request.

Our fourth requirement is to provide enough information in the models for the developer to give correct access rights to the right users. This also requires a direct mapping to HTTP requests and responses.

2.1.5 Domain Specific Requirements

A web service is designed with some specific service goals in mind. These service goals or requirements should be met by the service. The service requirements should be taken into consideration at the design phase to analyze at which point of service life cycle a particular requirement is fulfilled and allows for requirement traceability. Requirement traceability refers to the ability of the system to define, capture and follow the traces left by requirements on other elements of the software development environment [105]. We are interested in capturing the service requirements on the design models in order to design and develop REST web services with specific requirements in mind. This would also help us check which requirements are met by the service implementation and which not at the later stage of the service life cycle.

2.1.6 Time Requirements

Web services may have time critical behavior that must be taken into account when designing a web service. A web service cannot be trusted if it fulfills its service functionality but does not fulfill the time constraints imposed on it. This is because a web service is offered over the network and it may be used in another web service composition process. In such cases, if a service does not respond within a pre-defined time period, it can be assumed that the web service is not responding and may result in termination of the service. Thus, we require that timing constraints for the service should be modeled at the design phase such that the service can be validated in the later stages for its timed behavior as well.

2.2 Design Approach

Our design approach creates web services that are REST compliant. This means that the web services that are developed using our design methodology lead to web services that exhibit REST interface features and also address the requirements detailed in section 2.1.

The starting point of our approach is an informal web service specification in natural language that is used to build the resource and behavioral model of the web service. We use UML class diagram to represent the resource model and UML

protocol state machine with state invariants to represent the behavioral model of a REST web service. The resource model represents the resources of the service along with their data attributes, links between them and the different properties of these links.

The behavioral model of the service, represented by a UML protocol state machine with state invariant, define different states of the service and show how these service states change when its interface methods are invoked. The trigger methods in the behavioral model are restricted to HTTP methods PUT, POST and DELETE since these methods can make a change in the resource and GET method is used only to retrieve the state of the resource with no side-effects. We are able to define states of the service without compromising the requirements of stateless protocol by defining states as predicates over states of the resources. The representation of the resource, returned as a result of method invocation, is given by the attributes of resource defined in the resource model and the links that can be navigated further. These links are defined by observing the outgoing transitions from the target service state of the transition invoked by the interface method. The service requirements are also added in the behavioral model as labels and information of users with access rights to a method are added as actors with the transitions.

All these different types of information have mapping to HTTP request and response pairs. The Web Application Description Language (WADL) service descriptions can also be generated directly from the service design models. WADL is a machine readable format of REST web service interface [121].

The behavioral model also provides information about interface method contracts. A contract binds user of the service to pose a valid request and constrains its provider to provide the correct behavior. In our approach, we show how the pre conditions and the post-conditions of each service request can be generated from the proposed UML models and how these pre- and post conditions can lead to behavioral WADL interfaces.

We use as example an imaginary hotel room booking (HRB) service. The user of the service books a room and pays for it. While a third party service processes the payment, the service waits for the processing and marks the booking as paid once the confirmation is received. The booking can be canceled anytime if it is not waiting for the payment processing. The standard HTTP methods are called on the service to navigate through the different states of hotel booking service. Every piece of information that user can use, e.g., cancelation, payment and booking etc. is accessible via independent URIs. Also, information about when a method should or should not be invoked, e.g., making a booking cancel request, can also be inferred from the models. It is a simplified pedagogical example, but it shows how to design a REST interface for a service with a complex service state. In the next few chapters, we present in detail how the models containing all these informations are constructed.

2.3 Applications of Behavioral Interfaces

We have created behavioral interfaces of REST web services that provide pre and post conditions of the interface methods along with information on sequence of method invocations. A service description containing behavioral contracts has many applications that we describe below.

2.3.1 Code Generation

Service descriptions are often used to automatically generate code stubs to invoke the service from a particular programming language. The UML protocol state machines do not contain executable actions, unlike behavioral state machines, and hence are not executable. On the other hand, they provide behavioral information of an interface. Developers implementing a web service have to manually implement the interface specified in protocol state machine. This require efforts to ensure that implementation conforms to its behavioral specification. Our approach generates implementation stubs from the design models in an automated manner. In addition, behavioral specifications are automatically generated from the models and asserted as contracts into programmatic interface of the web service in Django web framework [66]. The service implementation can use the asserted contracts to validate a request from the client. The preconditions of a method provide a check on the incoming request. Thus, ensuring whether the conditions to invoke a service method are met before invoking the method can be an efficient activity in terms of cost and bandwidth. Similarly, a client can benefit from the asserted postconditions to validate a response from the server, constraining the provider of the service to ensure the functionality that is expected from it. Our approach to automatically generate code stubs in Django web framework from the design models is presented in Chapter 9.

2.3.2 Service Monitor

We can use the behavioral interface of a web service to provide a monitoring mechanism. The behavioral specifications can be added as a proxy interface to already developed and deployed web services to monitor their functioning. This helps in locating the fault in a service by observing the conditions that are not being met by the service methods. It can also check for any failure caused by a network fault and late delivery. Our work on implementing a service monitor in Django web framework [66] is also presented in Chapter 9.

2.3.3 Validation

The behavioral interface can be used to test a service implementation. In order to validate a service, test cases can be generated from the behavioral interfaces which can then be used to test the service implementation. The method contracts can

also be used for the generation of test oracles. Test oracles are used to determine whether a test has passed or failed. In the context of test case generation and test oracle generation, we can take advantage of several efforts done previously to validate the behavior of classes and web services using contracts [47] [45]. In Chapter 8, we present our model based validation approach for REST web service.

2.3.4 Specification for Developer

A web service developer can use the behavioral REST interface as a specification to implement the web service. Similarly, the users of a web service can use the models and contracts as detailed documentation on how to use a service correctly.

2.3.5 Publish Interfaces in the Standard Languages

The behavioral interface can be used to generate machine-processable syntactic interface. This facilitates in automatic service discovery and building of service repositories. Our approach provides direct mapping to WADL (Web Application Description Language) along with extensions to include method pre and post conditions. This is detailed in Chapter 5.

2.4 Related Work

In relation to our design approach to model RESTful behavioral interfaces, we find the work in the following two areas related to our work: role of contracts for web services and modeling REST web services.

2.4.1 Modeling REST web services

Many authors have investigated modeling of REST web services from different perspectives. We discuss these works in this section.

Schreier [115] presents a REST metamodel which is divided in structural and behavioral modeling. The resource types, their attributes, relations, interfaces and their representations are described for the structural modeling and the possibility of describing the behavior with state machine is also give. Compared to this work, our work uses standard UML metamodel to model REST web services. This allows us to use different developed tools and techniques that are already well adopted in the industry and are constantly improving due to large user base.

The work of Strauch and Schreier [120] presents a procedural model to transform a SOAP design to a RESTful HTTP design. The model uses a WSDL document of an existing SOAP service and refines it in three iterations into a RESTful interface. Kuuskeri et al. [76] present a detailed investigation on relationship between an actor computation model and the principles of REST. They provide a notation to apply the actor model to a RESTful web service and present it as

a network of actors. The aim of the paper is to provide better understanding of RESTful services. With the same aim of better understanding of RESTful services, Zuzak et al. [135] also present a generic model of RESTful systems based on nondeterministic finite-state machines with epsilon transitions. The model provides formalization of REST design principles including uniform interface, stateless client-server operation and code-on demand execution. Using these formal models, Zuzak and Shreier [136] give practical guidelines to design REST frameworks. In this work, they describe decomposition of client and server processing flows into generic modules which enabled better separation of concerns and improved system modifiability.

Ormeno et al. [97] present a proposal for modeling RESTful controllers using extended UML elements. Their work aims to focus on bridging the gap between RESTful modeling and implementation frameworks. To address this, they present a metamodeling process that facilitates an advanced designer to stereotype Java code and artifacts generated by Spring Roo. In comparison to their work, our work is not tied to any specific implementation framework and web services can be generated in different implementation technologies from models.

In [104], Perez et al. extend the OOh4RIA model-driven development process for Rich Internet Applications (RIAs) with a new model based on UML class diagrams. The class diagram maps the underlying server-side services to a RESTful interface consisting of resources, protocol actions and links. Their work focuses on static descriptions of resources and does not take into account the concept of application states and their tranformations.

Modeling of RESTful web services has also been addressed in the work of Markku et al. In [77, 78], Markku et al. present an approach that provides a step wise design transition from operation-centric view to data-centric view and provide model transformations in developing RESTful services and service APIs. They provide an approach that migrates legacy APIs to RESTful web services. This approach is further explored in [117]. They provide an iterative and incremental process for the development of model transformations by focusing on transforming information model to resource model. While their work talks about systematically transforming functional specifications into RESTful web services, our work addresses modeling of REST web services with a different perspective, i.e., developing REST web service for advanced scenarios.

2.4.2 Contracts and Web Services

The role of contracts in the domain of web services has also been investigated previously, e.g., [41] [34], etc. In [41], Castagna et al. present theory of contracts which formalizes the compatibility of a client to a service. They introduce a subcontract relation for behavioral typing of web services promoting service reuse or redefinition. In [33] and [34], a theory of contract is presented that addresses the problem of composition of multiple services. The correctness for service

compositions is modeled using process calculi and the notions of strong service compliance and strong subcontract pre-order are investigated. Contracts are also used in the work of Milanovic [92] for web service compositions. The work presents a contract-based approach to specify non-functional properties of a service with contract-based framework for service descriptions. The work is supported with enhanced directory capabilities, web service design patterns and verification of service compositions. In the context of modeling behavioral specifications and using contracts with UML, Lohmann et al. [86, 63] use visual contracts to specify the dynamic behavior and class diagrams to specify the static aspect of a web service. Graph transformations are annotated on to the class diagram with object diagrams specifying pre- and post-conditions of the operations.

In comparison to these works our work on generating contracts from the behavioral model does not require any additional design efforts. We extract pre- and post-conditions for service methods in an automated manner from models and instrument them in the code. The contract information is part of the behavioral interface developed to model stateful services.

2.5 Conclusion

REST web service interfaces have specific design requirements that must be taken into consideration during the design phase of web service development. A REST interface must provide the features of *connectivity*, *addressability*, *statelessness* and *uniform interface*. In addition, it should expose its functionality in its resources. In this chapter, we have identified and presented our design requirements in detail and have given an overview of our design approach create RESTful web services for advanced scenarios. The objective of our design approach is twofold. First, to provide a modeling approach that ensure that the designed interface follows the REST style. Second, to provide a way to describe behavioral services interface that specifies how to use a web service correctly and the expected results.

A behavioral REST interface has many applications. For example, a behavioral REST interface can be used to implement service monitor and can also facilitate the validation of service implementations. We can also directly generate code stubs or syntactic interfaces from the design models of the interface and these design models can also serve as the specification documents that can be referred to for understanding the service.

Chapter 3

Resource Models

The static structure of a service describes the basic entities that constitute it and the relationships among them. This static structure serves as a foundation for structuring and analyzing rest of the design of the service.

In this chapter we show how the static structure of a REST web service is defined using a UML class diagram. We show how the resources, their properties and the links among them are defined by imposing some additional constraints on the UML class diagram. This UML class diagram with additional constraints is called resource model. The resource model is built in accordance to the design requirements discussed in the previous chapter and it represents the properties of a REST web service interface. In section 3.1, we present resource model by first briefly explaining a UML class diagram, the concepts and properties of resources and how these concepts are mapped on a class diagram. The well-formedness rules for the resource model are inferred from these design decisions and presented in section 3.2. Section 3.3 concludes the chapter.

3.1 Resource Model

3.1.1 Class diagram

A UML class diagram represents the classes of a software and the associations between them. An association defines a relationship between two classes by which one class knows about the other class [125]. OMG specification defines class as a set of objects that share the same features, constraints and semantics [125]. An instance of class is called an object. An association specifies a semantic relationship between two classes or typed instances. It is usually represented with an arrow between two classes in the class diagram. The two ends can have role names. These roles are owned by the end class and specify the role that the class has to play in that association. The arrow may also have arrow head, i.e., a navigable end. The arrow head specifies that the association is navigable from the opposite end otherwise, the association is not navigable from the opposite end [125]. Association ends may

also be marked with multiplicity that specifies the number of class instances that can be associated in that association.

3.1.2 Resources

The concept of a *resource* is central to Resource Oriented Architecture (ROA). ROA is a structural design that fulfills design criteria presented by REST [111]. A resource is something that can be referred to and can have an address. Any important information in a service interface is exposed as a resource.

In REST, a resource is exposed via a URI and can be manipulated with standard HTTP methods. A resource can be either a collection resource or a normal resource. Collection resource does not have any attributes of its own and contains a list of other resources. A normal resource has its own attributes and represents a piece of information. The complexity of a service can be reduced by increasing the number of resources. This results in decoupling of information. The current state of the resource is given by the *representation* of the resource which is typically a document, e.g., an XML document or a JSON serialized object that contains information about the resource.

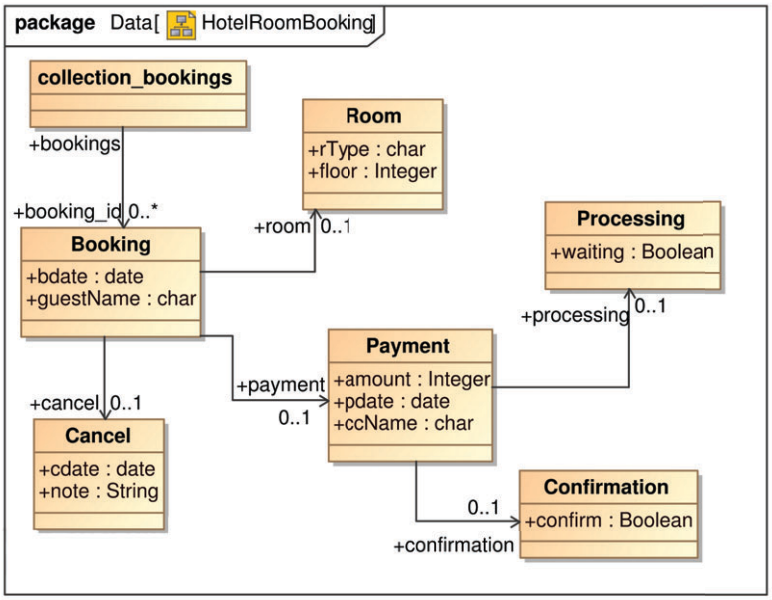
3.1.3 Modeling Resources

We are using UML class diagram with additional design constraints to represent resources, their properties and relation with each other. We have used the term *resource definition* to define resource entity such that its instances are called resources. This is analogous to the relationship between a *class* and its *objects* in object oriented paradigm.

In our resource model, we represent *resource definitions* as classes. A collection *resource definition* is represented by a class with no attributes and a normal *resource definition* has one or more attributes. Each association has a name and minimum and maximum cardinalities. These cardinalities define the minimum and maximum number of resources that can be part of the association.

We also define a *root resource definition* in the resource model which is typically a collection resource with no incoming associations. The *root resource definition* provides the starting point for the navigation path to all the other resources. A resource model can have more than one *root resource definition*. In such a case, the resources can have more than one addressable path. However, one addressable path should belong to only one resource.

We use the example of an imaginary hotel room booking (HRB) service to describe our resource model. The hotel room booking service, explained in the previous chapter, allows the user of the service to book a room, pay for the reservation, and cancel it. Figure 9.1 shows the resource model of the hotel room booking RESTful service. The hotel room booking service is composed of




```
/{booking_id}/
/{booking_id}/cancel/
/{booking_id}/payment/
/{booking_id}/room/
/{booking_id}/payment/processing/
/{booking_id}/payment/confirmation/
```

Figure 3.1: (Top) Resource Model for HRB RESTful Web Service. (Bottom) Resource paths

one collection *resource definition* (*collection_bookings*) and six normal *resource definition* (*Booking, Room, Payment, Processing, Confirmation, and Cancel*).

3.1.4 Mapping Resources to Class Diagrams

A direct mapping between elements of a class diagram and the concepts of ROA is as under.

A *resource definition* is represented by a class.
A resource is an instance of a *resource definition*, analogous to the object of a class.
A collection *resource definition* is represented by classes that have no attributes and their name starts with *collection_*. It has one outgoing transition with multiplicity of 0...* for the contained *resource definition* indicating that a collection resource can have none or many resources.

- A *root resource definition* is a collection *resource definition* with no incoming edges.
- The data of resource representation is mapped to the attributes of a class.
- The connectivity between the resource definitions is represented by the associations.
- The role names on the association ends give the relative URI addressees of the resources.
- The number of resources that can be take part in an association is defined by the multiplicity constraints on the association.

We require that every association must have a role name in order to form URI addresses. The attributes of classes must be public since the representation of a resource is available for manipulation and they must have a type since they represent a document containing information of the resource.

Some pieces of information can be attributes of classes representing *resource definitions*. In such cases, if the attribute value is *False/NULL* they return no information and if the attribute has some value or it is *True*, they return its representation. In our resource model, we have separated model attributes into *resource definitions* in order to allow manipulation of their values separately and create their URI addresses.

3.1.5 Addressability

Addressability requires that every piece of information is addressable via a URI.

In a resource model, the URI of a *resource definition*, *r*, is obtained by traversing the path formed by the successive associations from the *root resource definition* to *r*. The role names on the association ends constitute the URI address. In Figure 9.1, *collection_bookings* is a *root resource definition* and the paths on the bottom of Figure 9.1 are valid. A GET method on a collection *resource definition* returns a list of all the resources it contains, if any. Similarly, a GET on a *resource definition* will return the resource representation if it exists or no information if it does not exist. For example, if a *Payment* resource exists then a GET method on *booking_id*/*payment*/ will give the representation of *Payment* containing its details in a JSON or XML document.

The REST style requires that all the *resource definitions* should be addressable. In our context this requirement is fulfilled if each *resource definition* can be reached from the *root resource definition* with at least one path by navigating one or more associations. The paths visiting the same association more than once are not valid.

3.1.6 Methods

A UML class diagram allows us to define a number of operations for each class. However, in a RESTful interface, resources do not have different access methods, instead the standard HTTP methods are used. This property leads to a *uniform*

interface since all classes would have only from one to four method names, i.e., GET, POST, PUT and DELETE. A GET method can be invoked on every resource to retrieve the current state of the resource. The information on the allowed side-effect methods on a resource, i.e., PUT, POST and DELETE is inferred from the behavioral model, hence we do not consider necessary to add this information in the resource model.

3.1.7 Connectedness

The links between resources connect the resource definitions and provide connectivity to the resource model. These links are represented as association between classes. For the service to be fully compliant with REST architectural style, the resource representations should also contain the list of links that can be taken further in order to emulate the stateful behavior of REST web service. This information is obtained by traversing the outgoing transitions of the target state in the behavioral model. This is explained further in Chapter 4.

3.2 Well-formedness Rules for Resource Model

We have imposed some design requirements on a UML class diagram that must be taken into consideration by the developer when constructing models for the REST web service. These design decisions are explained in the last section. Below, we present a list of these design decisions as well-formedness rules for resource model

- The class name representing collection *resource definition* should start with *collection_*
- The resource model should have atleast one class representing collection *resource definition* that is considered as a *root*.
- The graph formed by classes and associations should be connected.
- Associations should have role names on the association ends.
- Each class should have a navigable path association from the *root*.
- Classes should not contain methods.
- Class attributes should have a type.
- Class attributes should be public.

3.3 Conclusion

In this chapter, we present the static structure of a REST web service as a resource model. The resource model is a UML class diagram with additional constraints to represent different types of resources, their properties and links with each other. It is designed with the intention to cover the REST interface requirements. The resource model provides addressability feature by constraining the associations to have role names. Resources are navigated through these associations, thus,

providing addressability. The resource graph is required to be connected and does not contain information on methods since a REST web service provides a uniform interface. With the help of an example of a hotel room booking service, we demonstrate how the REST interface requirements are met by our resource model. We also present a list of well-formedness rules for a resource model that should be followed by the developer in order to create REST interfaces.

Chapter 4

Behavioral Models

The purpose of the behavioral model is to describe the dynamic structure of behavioral interface of a RESTful web service. A behavioral interface specifies the order of method invocations and method contracts in order to obtain the desired goals of the service. These method contracts give the preconditions and postconditions of method calls.

We propose to use a UML protocol state machine with state invariants to describe the allowed operations in a web service. We consider that a UML protocol state machine is suitable for representing the behavior of a web service interface as it provides interface specifications without actions or execution details and contains information on conditions under which the methods can be invoked and the expected output from them.

In this chapter, we show how the dynamic structure of behavioral interface of a REST web service is modeled with UML protocol state machine. We start with a detailed introduction of UML protocol state machine and motivate the need to generate behavioral information from it. In section 4.2 we use formal definitions and small examples to show how a behavioral interface can be generated for a class. We apply this approach first on a class and its protocol state machine in order to focus on the conceptual underpinnings of the contract generation approach without the design constraints introduced for the REST models. The REST behavioral model is then presented in Section 4.3. The concept of synchronous and asynchronous web services is presented in Section 4.4. The representation for authorized actors and domain-specific requirements in behavioral model is shown in Section 4.5 and 4.6, respectively. The time constraints are discussed in Section 4.7.. The notion of supporting stateful behavior of REST services with a stateless protocol in our model is discussed in Section 4.8. Based on the design decisions, we give the well-formedness rules for our behavioral model in Section 4.9. The chapter is concluded in section 4.10.

4.1 Protocol State Machines and Class Contracts

State charts are one of the UML behavioral diagrams. They represent behavior of model elements with finite state transition systems. State charts were initially adopted in UML 1.3 as a variant of David Harel's state charts [62]. In UML 2.2, state charts are adopted as state machines as an effort to separate the semantics of activity diagrams from state machines [125].

There are two types of state machines in the current version of the UML standard: *behavioral state machines* and *protocol state machines*[51]. Behavioral state machines specify how an object reacts to a sequence of events. The effect of a transition is specified in an action, usually defined as an executable statement in a programming language. On the other hand, a protocol state machine describes (part of) an interface specification. In protocol state machine transitions are triggered by call events (invoking an operation), and the behavior is specified by using transition pre- and post-conditions. We consider behavioral state machines to be more suitable to describe reactive behavior while protocol state machines are more suitable to describe classes that combine data and stateful behavior.

It is possible to generate executable code from behavioral state machines, since the transitions include executable actions. The implementation of behavioral state machines has been discussed often in the literature [95, 26, 28] and there are commercial tools that provide automatic code generation from behavioral state charts such as Telelogic Rhapsody [60].

In contrast, protocol state machine does not include executable actions, but only a specification of these actions in the form of preconditions and postconditions. This approach is useful to describe the interface and intended behavior of a class, while omitting its implementation details. In this case, the protocol state machine serves as a visual representation of a behavioral interface of a class. A programmer who plans to use a class can inspect its protocol state machine to know what methods are available, when these methods can be invoked and what the expected results are.

Since a protocol state machine does not include executable actions, the actual implementation of a protocol state machine into a class in a programming language such as Java has to be performed manually. This requires efforts to ensure that a class implementing a protocol state machine behaves as described in its interface. We address this requirement by inferring information from protocol state machine and asserting it as contract in the class implementation.

The use of class contract to specify the behavior of a software class has been advocated by Meyer [91] and implemented first in the Eiffel programming language. More recently, the Java Modeling Language (JML) [83] provides a contract language to annotate Java classes. Also, the LIME specification language [80] is an approach that allows us to annotate Java program with pre- and post-conditions as well as with propositional linear temporal logic.

Thus, we can generate class contracts from UML protocol state machine and

instrument them in the code. This information can then be used to validate the behavior of the class in later stage of the development cycle.

4.2 Generating Behavioral Interface

In this section, we demonstrate how class contracts can be generated from protocol state machine. In order to generate a class contract we require two UML diagrams: a UML class diagram and a UML protocol state machine. The UML class diagram provides a syntactic interface of a class by naming the public methods of a class and the type of their input and output parameters. The protocol state machine describes the behavioral interface of a class.

The protocol state machine should define a state invariant for each simple and composite state. A state invariant is a boolean expression that is true when the given state is active. This expression should be pure or free of side effects. State invariants link the current state of an object, defined using object attributes and the current state of its protocol, defined using a protocol state machine. We require that a state invariant is defined using public features of the class interface. That is, the state invariant should be observable by other objects.

As an example, Figure 4.1 shows a syntactic interface for a bounded stack class and a simple protocol state machine with three simple states ($Empty, notEmpty$ and $Full$), four query methods ($isEmpty(), isFull(), getMax()$ and $size()$) and two operations ($push()$ and $pop()$). The state invariants $isEmpty()$, *not* $isEmpty()$ *and not* $isFull()$ and $isFull()$ must be true for the states *Empty*, *notEmpty* and *Full*, respectively. Transitions are triggered by the *push(o)* and *pop()* operations with associated guard conditions.

Our objective is to extract a class contract for the class *Stack* so that it can be expressed in a contract language such as JML or LIME. This contract should follow the protocol defined in the state machine. That is, the observable behavior of a class that implements the protocol state machine and a class that implements the contract should be equivalent. The contract constrains the users of a class. For example, the method *push()* cannot be invoked if *isFull()* evaluates to true. It also constrains the implementation of a class. For example, after invoking *push()*, the expression *not isEmpty()* should evaluate to true.

We should note that while there are three different transitions triggered by the $push()$ method, the actual implementation should combine the behavior of the three transitions into one method. Therefore, in order to generate the class contract we need to combine the information stated in all the transitions triggered by a method into a precondition and postcondition for that method. We describe this task in the following sections.

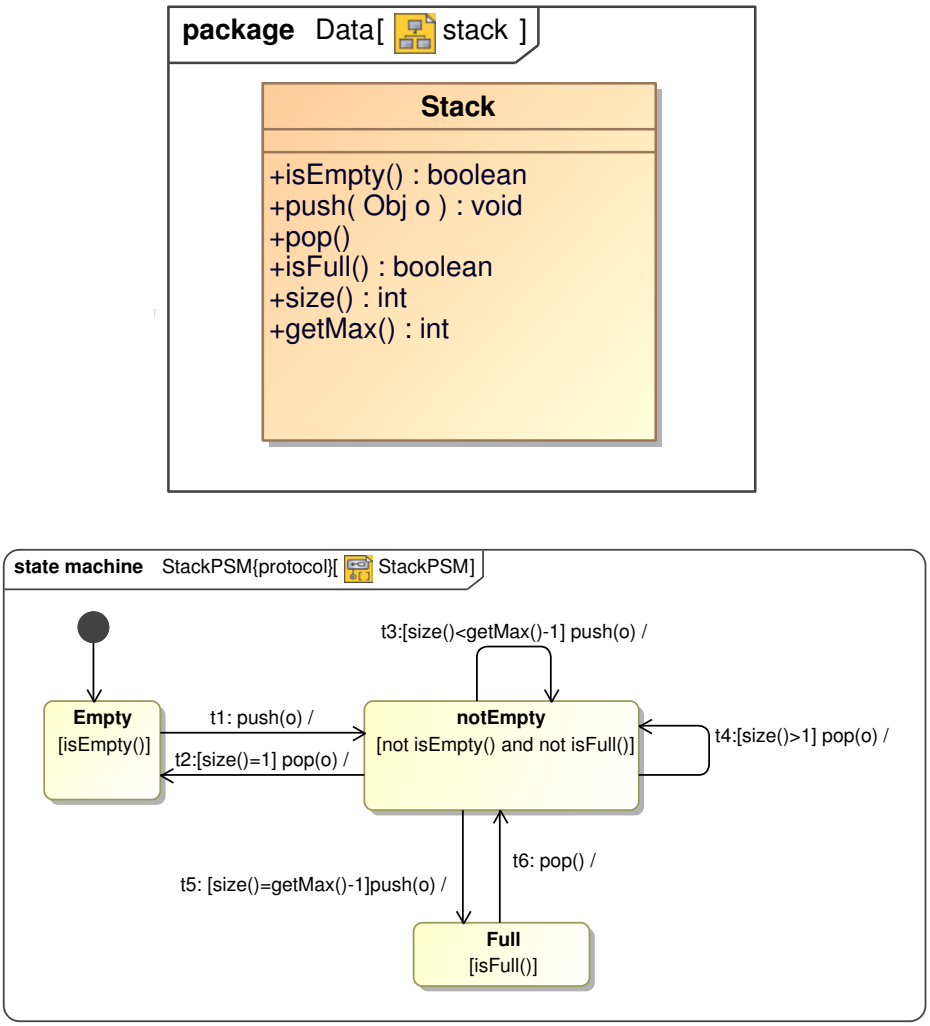


Figure 4.1: (Top) Stack Class. (Bottom) Protocol State machine of Stack Class

4.2.1 Defining the Structure of Protocol State Machines

In this section we study the structure of a protocol state machine as described in the UML standard [125].

There are three kinds of states i.e., simple, composite and submachine state [125]. We do not define a submachine state since it is considered semantically equivalent to a composite state [125].

Each state has a state invariant which is a boolean function. The substates of a composite state cannot have invariants that would weaken the invariant of the parent state. As an example consider a composite state with the invariant $x > 0$. A substate with an invariant stating $x = 0$ would cause an error as it is not in the scope of $x > 0$. Instead, a substate with the invariant $x > 1$ is appropriate as it is in the scope of $x > 0$ and would further restrict the parent state invariant.

A transition is a directed relationship between the two states represented by an arrow from a source state to a target state [125]. The trigger that fires a transition is annotated on the arrow alongwith the pre- and post-conditions as `[precondition]trigger/[postcondition]`.

Based on this, we can now define the structure of a protocol state machine.

Definition 1 A protocol state machine is defined as a set such that: $pSM = \{ S_F, \iota, S_s, S_c, T, \ source, target, \ trigger, issubstate, g, inv, reg, post \}$, where S_F is a set of final states, ι is the initial state, S_s is a set of simple states, S_c is a set of composite states and T is a set of transitions. The following functions describe the relations between the elements of the protocol state machine:

- $issubstate(s_1, s_2)$ is true if state s_1 is a substate of s_2. We require that the graph created by the set of states S and the binary relation $issubstate$ is connected and acyclic.
- $trigger(t)$ provides the operation that triggers the transition t.
- $source(t)$ is the set of source states for a transition t. This is the set of all states that should be active in order to trigger the transition t. If a transition starts from a substate, then this set will also include all the containing states. If a transition is a join transition, then this set will include all the states participating in the join (and their containers). We call the direct source of a transition to the set of source states without their containers: $dsource(t) = \{s \in source(t) : \neg \exists s' \in source(t) : issubstate(s', s)\}$.
- $target(t)$ provides the set of all the target states for a transition t.
- $post(t, \sigma)$ evaluates the postcondition of the operation associated to a transition t in an object in state σ.
- $g(t, \sigma)$ evaluates the guard associated with the transition t in an object in state σ.
- $reg(s, \sigma)$ returns the region to which a state s belongs to in an object in state σ.
- $inv(s, \sigma)$ evaluates the invariant of the state s in an object in state σ

The set of states in a protocol state machine is the union of initial state, final states, simple states and composite states, i.e., $S = \iota \cup S_F \cup S_s \cup S_c$. The sets S_s and S_c are mutually disjoint. Elements in the set of final states can never be the source of a transition, i.e., $S_F = \{s : \forall t \in T, s \notin source(t)\}$. An initial state cannot belong to the set of final states and is not the target state of any transition, i.e., $\iota = \iota \in S \wedge \iota \notin S_F \wedge \forall t \in T : \iota \notin target(t)$

The structure of a protocol state machine ensures that if a state is not contained in any other state, i.e., it is at the highest level of state hierarchy then only one state can be active at the same time. If the active state is a substate then no other state can be active in the same region. We can formalize this with the following condition:

$wellformed(\sigma) = \exists s \in S : inv(s,\sigma) = \forall s' \in S : \neg issubstate(s,s') \implies \neg\exists s'' \in S : inv(s'',\sigma) \wedge (\exists s' \in S : issubstate(s,s') \implies \neg\exists s'' \in S : inv(s'',\sigma) \wedge reg(s,\sigma) = reg(s'',\sigma))$

4.2.2 Semantics of Protocol State Machines

The structure of protocol state machine described above provides a concrete basis to describe the semantics of protocol state machine as a state transition system. The behavior of invoking a method will be equivalent to triggering one or more transitions that have that method as a trigger. The set of transitions triggered simultaneously is called a *step*. To define the semantics of a protocol state machine we need to define what transitions are triggered in a *step* and what the effect of triggering each transition is.

A state invariant can be used to define a pre- or post-condition of a transition. When a transition points to a state, this specifies that the transition must not fire if the state invariant of the target state does not hold afterwards. With this in mind we can think of the state invariant as a postcondition. Analogous to this, the state invariant of the source state of a source state can act as a precondition. If that invariant does not hold, it means we are not in the source state and the transition can not be fired.

In order to trigger a transition it should be enabled. A transition is enabled when all its source states are active, the guard of the transition is true and the trigger of the transition matches the method invoked. If no guard is given for a transition, then we assume it is true. Similarly, if the postcondition of the transition is not specified, it is assumed to be true.

Definition 2 A transition t is enabled if $enabled(t,m,\sigma) = g(t,\sigma) \wedge (trigger(t) = m) \wedge \forall s \in source(t) : inv(s,\sigma)$

It is possible that invoking a method triggers more than one enabled transitions.

In some cases we can trigger only one of the enabled transitions. This is when two enabled transitions, with the same trigger, try to exit one or more common states and target different states. This situation results in a conflict between the two transitions that are enabled.

Definition 3 Two transitions t_1 and t_2 are in conflict when:
$conflict(t_1,t_2) = (trigger(t_1) = trigger(t_2)) \wedge enabled(t_1,trigger(t_1),\sigma)$
$\wedge enabled(t_2,trigger(t_2),\sigma) \wedge (source(t_1) \cap source(t_2) \neq \emptyset)$

When two transitions are in conflict we need to choose one transition to fire. The UML standard defines a priority scheme based on the state hierarchy. Transitions originating from a deeper substate has priority over transitions originating from a composite state.

Definition 4 Transition t_1 has priority over transition t_2 when $priority(t_1,t_2) =$
$conflict(t_1,t_2) \wedge \exists s_1 \in dsource(t_1), s_2 \in dsource(t_2) : issubstate(s_1,s_2)$

When a method is invoked in an object whose behavior is represented as a protocol state machine, the behavior of the method should be equivalent to triggering all the transitions in a *step* of the protocol state machine.

The set of transitions triggered simultaneously is called a *step*. We can now define what a *step* in a protocol state machine is.

Definition 5 Given a protocol state machine, an object in state σ and a method m, we define a *step* as the set of all enabled prioritized transitions:
$step(m,\sigma) = \{t \in T : enabled(t,m,\sigma) \wedge (\neg \exists t' \in T : enabled(t',m,\sigma)$
$\wedge priority(t',t))\}$.

In some cases, transitions may be in conflict but none of them may have priority over the other. This is when more than one transitions are triggered by the same method call, originate from same direct source state and target different states. In this case, the behavior of a protocol state machine is non-deterministic. The UML standard specifies that in case of non-determinism between two or more transitions, any of the transitions can be triggered.

The set of transitions that are in conflict with each other in a *step* set in state σ is given as follows.

Definition 6 A conflict set is a set of all the conflicting transitions in the *step* set with transition t in state σ:
$SConflict(t,\sigma) = \{t' \in step(trigger(t),\sigma) : conflict(t,t')\}$

We deal with the case of non-determinism in the next section while generating the postcondition of such transitions.

The effect of firing a transition is such that invariants of all the target states should be true and the transition postcondition should be also true.

Definition 7 The effect of triggering a transition t is defined as:
$effect(t, \sigma') = post(t, \sigma') \wedge \forall s \in target(t) : inv(s, \sigma')$

This also caters to the case of a fork transition by ensuring that invariants of all the target states are true.

4.2.3 Generation of Class Contract

The precondition of a method states under which conditions we can invoke a method. We allow a method to be invoked in a state σ when it can trigger at least one transition in the equivalent protocol state machine. This is the case when there is at least one transition enabled.

Definition 8 The precondition for a method m is defined as:
$precondition(m, \sigma) = \exists t \in T : enabled(t, m, \sigma)$

Since the structure of a protocol state machine is finite (there is a finite number of states, transitions, triggers) and static (it does not change at runtime), we can replace the existential quantification in the previous definition with a disjunction. In our example of *Stack*, we can calculate the precondition for the operation $push()$:

$precondition(push, \sigma) = enabled(t1, push, \sigma) \vee enabled(t3, push, \sigma) \vee$
$enabled(t5, push, \sigma)$

By replacing the definition of *enabled* for each transition we obtain the following precondition for $push()$:

$precondition(push, \sigma) = isEmpty(\sigma) \vee (size(\sigma) < getMax(\sigma) - 1 \wedge \neg isEmpty(\sigma)$
$\wedge \neg isFull(\sigma)) \vee (size(\sigma) = getMax(\sigma) - 1 \quad \wedge \neg isEmpty(\sigma) \wedge \neg isFull(\sigma))$

We should note that this expanded precondition does not refer anymore to the structure of protocol state machine. It uses transition guards and state invariants, but as we stated in Section 4.3, we require that the guards and invariants are defined in terms of public features of the class. Thus, we have extracted a method precondition using information from the protocol state machine that can be represented in existing class contract languages such as Eiffel or JML, or even as assertions in languages such as Java or C++.

In a language such as JML, the previous precondition can be represented as:

```
/*@ requires (isEmpty()) ||
  @          (!isEmpty() && !isFull() && size()<getMax()-1) ||
  @          (!isEmpty() && !isFull() && size()==getMax()-1) */
void push(Object o) { ...
```

The postcondition of a method states the outcome of invoking the method. If the transitions in the *step* set are in conflict i.e., causing a nondeterministic

behavior, then any one of the transitions can be fired such that the structure of protocol state machine is not violated. If there is no conflict between the enabled transitions then all the transitions of the $step$ set are triggered and the effect of each triggered transition should be observable after executing the method. We represent as σ the state of an object before executing the method and as σ' the state of the object after executing the method.

Definition 9 The postcondition for a method m is defined as:
$postcondition(m,\sigma,\sigma') = \forall t \in step(m,\sigma) : (SConflict(t,\sigma) \neq \emptyset \implies$
$\exists t' \in SConflict(t,\sigma) : \neg effect(t,\sigma) \implies effect(t',\sigma)) \wedge (SConflict(t,\sigma) = \emptyset$
$\implies effect(t,\sigma'))$

This implies that if the set of conflicting transitions is not empty, then any of the enabled transitions can be triggered. The definition ensures the well formedness property of protocol state machine by allowing the effect of any other transition to be true only if the effect of transition under check is false.

In addition, our definition of post-condition also caters to the case of self-transition, i.e., a transition that has the same source and target states. A definition that negates the invariant of the source state of a transition would be too strong to address the case of self-transition.

The universal quantification over the $step$ set can be replaced by the conjunction of three implications, in our *Stack* example, as the structure of protocol state machine is finite and static.

$postcondition(push,\sigma,\sigma') = (enabled(t1,push,\sigma) \implies effect(t1,\sigma')) \wedge$
$(enabled(t3,push,\sigma) \implies effect(t3,\sigma')) \wedge (enabled(t5,push,\sigma) \implies$
$effect(t5,\sigma'))$

When we replace the definitions of $enabled$ and $effect$, we obtain:

$postcondition(push,\sigma,\sigma') = (isEmpty(\sigma) \implies \neg isEmpty(\sigma') \wedge \neg isFull(\sigma')) \wedge$
$(size(\sigma) < getMax(\sigma) - 1 \wedge \neg isEmpty(\sigma) \wedge \neg isFull(\sigma) \implies \neg isEmpty(\sigma') \wedge$
$\neg isFull(\sigma')) \wedge (size(\sigma) = getMax(\sigma) - 1 \wedge \neg isEmpty(\sigma) \wedge \neg isFull(\sigma) \implies$
$isFull(\sigma'))$

In the JML contract language the previous value of an expression (before a method is executed) can be obtained by using the `\old` clause. As an example, this is the postcondition of the push operation:

```
/*@ requires(isEmpty()) ||
  @         (!isEmpty() && !isFull() && size()<getMax()-1) || (!
      isEmpty() && !isFull() && size()==getMax()-1) */
  @ ensures (\old(isEmpty()) ==> !isEmpty() && !isFull()) &&
  @         (\old(!isEmpty() && !isFull() && size()<getMax()-1) ==> !
      isEmpty() && !isFull()) &&
```

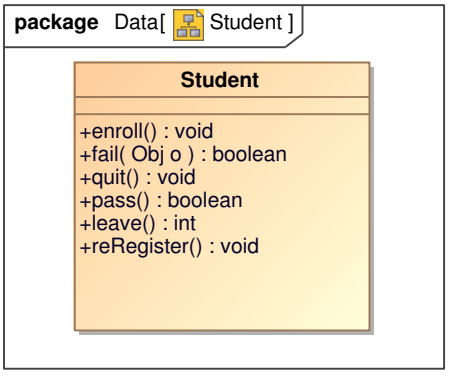


Figure 4.2: Student Class

```
  @          (\old(!isEmpty() && !isFull() && size()=getMax()-1) ==>
       isFull()) */
  void push(Object o) { ...
```

The generation of the precondition and postcondition of the *pop* operation follows a similar pattern.

Usually the main drawback of using postconditions in testing and runtime checking is that it needs to refer to the previous state of the object before executing the method. This can be achieved at runtime by storing a snapshot of the object, but this step can be computationally expensive and it is only partially supported in contract languages such as Eiffel and JML. However, a closer inspection of Definition 2 of *enabled* reveals that we do not need to store the complete state of an object but only the guards and invariants that are enabled. Usually, that only requires few bits of storage per method.

4.2.4 Example

In order to see the effectiveness of our approach, we use a *Student* class and extract contracts from its protocol state machine. The *Student* class and its protocol state machine is shown in Figure 4.2 and Figure 4.3, respectively. According to Figure 4.3, a registered student can enroll in a course and start studying. During the course, he has to attend labs and give exams. If he fails in the midterm, he has the option to withdraw the course and wait for the next course. A student also has the option to quit school anytime he wants to while studying. If he clears his labs and exams, he is passed, else he can register either in the current semester or wait for another semester.

We use different types of transitions in our example to show the applicability of our approach. These include fork transition, join transition, high-level transition and self transition. We also deal with the case of conflicting transitions where

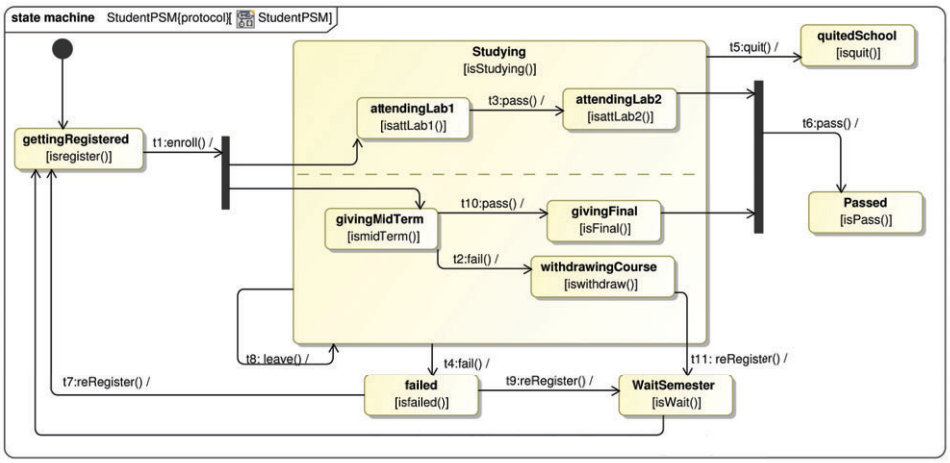


Figure 4.3: Protocol State machine of Student Class

either one transition has priority over the other or they cause a non-deterministic behavior of the protocol state machine. Below, we explain these cases separately.

Fork Transition

Transition $t1$, in Figure 4.3, is a fork transition with one source state and more than one target states. These target states are triggered simultaneously by the method $enroll()$ in state $gettingRegistered$. The precondition for the method $enroll()$ involves those transitions that are enabled in state .

$$precondition(enroll\quad) = enabled(t_1\ enroll\quad)$$
$$precondition(enroll\quad) = isregister(\quad)$$

The $step$ set will contain all the transitions that are enabled in the state . As $t1$ has no conflicting transitions, so the post-condition for $SConflict = \emptyset$ is the effect of all the transitions that are enabled in the $step$ set.

$$postcondition(enroll\quad) = enabled(t1\ enroll\quad) = effect(t1\quad)$$
$$postcondition(enroll\quad) = isregister(\quad) = (isStudying(\quad)\ isattLab1(\quad))$$
$$(isStudying(\quad)\ ismidTerm(\quad))$$

Definition 7 ensures that all the target states of the fork transition $t1$ are true as a postcondition.

Join Transition

Figure 4.3 shows that invoking method *pass*() triggers either transition $t3$, $t10$ or $t6$ in states $attendingLab1$, $givingMidTerm$ or in the orthogonal states $attendingLab2$ and $givingFinal$. Transition $t6$ is a join transition with one target state and more than one source states in orthogonal regions. According to Definition 2, the invariants of all the source states of $t6$ should be true before firing $t6$. The precondition for method *pass*() is given as:

$precondition(pass, \sigma) = (isStudying(\sigma) \wedge isattLab1(\sigma)) \vee (isStudying(\sigma) \wedge ismidTerm(\sigma)) \vee (isStudying(\sigma) \wedge isattLab2(\sigma) \wedge isFinal(\sigma))$

All the three enabled transitions in state σ i.e., $t3$, $t10$ and $t6$ do not have conflict with any other transition, so according to Definition 9 the postcondition of *pass*() is extracted as their $effect$.

$postcondition(pass, \sigma, \sigma') =$
$((isStudying(\sigma) \wedge isattLab1(\sigma)) \implies (isattLab2(\sigma') \wedge isStudying(\sigma))) \vee$
$((isStudying(\sigma) \wedge ismidTerm(\sigma)) \implies (isFinal(\sigma) \wedge isStudying(\sigma'))) \vee$
$((isStudying(\sigma) \wedge isattLab2(\sigma) \wedge (isStudying(\sigma) \wedge isFinal(\sigma))) \implies$
$isPass(\sigma'))$

High Level Transition

The high level transition $t5$ originates from a composite state and results in exiting all the substates of *Studying* whenever *quit*() method is invoked. The pre-condition for *quit*() method is given as:

$precondition(quit, \sigma) = isStudying(\sigma)$

This transition is triggered whenever the method *quit*() is called and invariant of the composite state *Studying* is true i.e., protocol state machine is in any of the substate of *Studying*.
The postcondition for *quit*() is given as the $effect$ of $t5$ since:
$SConflict(t5, \sigma) = \emptyset$,

Thus,
$postcondition(quit, \sigma, \sigma') = isStudying(\sigma) \implies isquit(\sigma')$

Self Transition

Transition $t8$, triggered by $leave()$, is a self transition as it has the same source and target states. It is triggered whenever composite state *Studying* is active and *leave* method is called.

$precondition(leave, \sigma) = isStudying(\sigma)$

Since, $t8$ is a self-transition on a composite state *Studying*, so it can be fired from any of its sub states. With an empty conflict set of $t8$, we get:

$postcondition(leave, \sigma, \sigma') = enabled(t8, leave, \sigma) \implies effect(t8, \sigma') = isStudying(\sigma) \implies isStudying(\sigma')$

A self transition on a composite state implies that same sub-state should be active after the transition that was active before firing the transition. We consider our definition as a weak case of self transition since it does not cater to the case of sub-states currently. A stronger definition of post-condition would ensure that the same substate is active after the self transition that was active before it was fired.

Conflicting Transitions

As defined in Definition 4, two transitions are in a conflict when they are both enabled, triggered by the same method and have at least one common source state. In Figure 4.3, transitions $t2$ and $t4$ are in conflict since they both have the same trigger method $fail()$, they are both enabled in state *ismidTerm* and they both have a common source state i.e., *Studying*. Similarly, $t7$ and $t9$ are invoked by the same operation i.e. $reRegister()$ and they leave the same source state i.e., $failed$. $t11$ also has the same operation as $t7$ and $t9$,i.e. $reRegister()$ but it is not in conflict since it has no common source state with them.

In both the cases, only one of these conflicting transitions can be fired. This problem can be resolved by using UML 2.0 well-formedness rules. The conflicting transitions can be such that one transition has priority over the other or they can cause a non-deterministic behavior of the protocol state machine. We explain both these cases with this example below.

Case of Priority Transition

According to UML standards, a conflicting transition may have priority over the other if the direct source of one of the transitions is a deeper state compared to the direct source of other transition which is a composite state. The precondition of $fail()$ in state σ requires that either $t2$ is enabled or $t4$ is enabled.

$precondition(fail, \sigma) = (isStudying(\sigma) \wedge ismidTerm(\sigma)) \vee isStudying(\sigma)$

As the direct source of transition $t2$ is a substate of the direct source of transition $t4$ in Figure 4.3, so according to Definition 4 $t2$ has priority over $t4$ in state *givingMidTerm*. Therefore, only $t2$ is defined as an enabled prioritized transition

in $step$ set in state σ and its effect is calculated as a post-condition.

$$postcondition(fail, \sigma, \sigma') = ismidTerm(\sigma) \wedge isStudying(\sigma) \implies iswithdraw(\sigma') \wedge isStudying(\sigma')$$

For all the other sub-states of composite state *Studying*, $t4$ is fired as a high-level transition with no conflicting transitions.

$$postcondition(fail, \sigma, \sigma') = isStudying(\sigma) \implies isfailed(\sigma')$$

Taking the conjunction of both the implications, the postcondition of method $fail()$ is given as follows:

$$postcondition(fail, \sigma, \sigma') = ((ismidTerm(\sigma) \wedge isStudying(\sigma)) \implies iswithdraw(\sigma') \wedge isStudying(\sigma')) \wedge ((isStudying(\sigma)) \implies isfailed(\sigma'))$$

Case of Non-Deterministic Behavior

When the operation $reRegister()$ is invoked both $t7$ and $t9$ are enabled in state $failed$. This causes a non-deterministic behavior of the protocol state machine since none of the transitions has priority over the other.

$$precondition(reRegister, \sigma) = isfailed(\sigma) \vee isfailed(\sigma)$$

Both the enabled transitions result in a non-empty conflict set in state $failed$ of object of class *Student*.

$$SConflict(t7, \sigma) = \{t9\}$$
$$SConflict(t9, \sigma) = \{t7\}$$

The well-formedness rule of UML 2.0 for non-deterministic behavior is given in Definition 9. It implies that in the case of conflicting enabled transitions, any of the conflicting transitions can be fired such that the structure of protocol state machine is not violated. Thus,

$$postcondition(reRegister, \sigma, \sigma') = (isfailed(\sigma) \implies (\neg isWait(\sigma') \implies isregister(\sigma'))) \wedge (isfailed(\sigma) \implies (\neg isregister(\sigma') \implies isWait(\sigma')))$$

Listing 4.1 shows the interface of *Student* class annotated with JML contracts as explained in this section.

Listing 4.1: JML specifications for Student Protocol state machine

```
/*@ requires (isregister()) */
  @ ensures  (\old(isregister()) ==> ((isStudying() && isattLab1()) &&
  @                                   (isStudying() && ismidTerm())))
      */
void enroll(Object o) { ...

/*@ requires ((isStudying() && isattLab1()) || (isStudying() &&
    ismidTerm())
  @          || ((isStudying() && isattLab2()) && (isStudying() &&
      isFinal()))   */
  @ ensures ((\old(isStudying() && isattLab1())) ==> (isStudying() &&
      isattLab2())) ||
  @         (\old(isStudying() && ismidTerm()) ==> (isStudying() &&
      isFinal())) ||
  @         (\old((isStudying() && isattLab2()) && (isStudying() &&
      isFinal()))
  @          ==> (isPass())) */
void pass(Object o) { ...

/*@ requires (isStudying()) */
  @ ensures  (\old(isStudying()) ==> isquit()) */
void quit(Object o) { ...

/*@ requires (isStudying()) */
  @ ensures  (\old(isStudying()) ==> isStudying()) */
void leave(Object o) { ...

/*@ requires (isStudying() && ismidTerm()) || (isStudying()) */
  @ ensures  (\old(isStudying() && ismidTerm()) ==> iswithdraw() &&
      isStudying()) && (\old(isStudying()) ==> isfailed()) */

void fail(Object o) { ...

/*@ requires (isfailed() || isfailed() || (isStudying && iswithdraw())
    ) */
  @ ensures  (\old(isfailed()) ==> ((isWait() && !isregister()) || (!
      isWait() && isregister())))  && (\old(isfailed()) ==> ((
      isregister() && !isWait()) || (!isregister() && isWait() ))) &&
      (\old(isStudying && iswithdraw()) ==> (isWait() && !isregister()
      ))*/
void reRegister(Object o) { ...
```

We have applied this approach to generate contract from UML protocol state machine on our behavioral REST model. It is addressed in detail in the next chapter in section 5.1. However, before understanding how contract information is inferred from the REST behavioral model, we need to understand its structure. This is explained in the next section.

4.3 Behavioral Model

The behavioral model of a REST web service defines its dynamic structure. It explains the different service states of a REST service during its life cycle and how those states are traversed. We use UML protocol state machine with state invariants and additional design constraints to represent the REST behavioral model.

In chapter 3, we showed how the static structure of a hotel room booking REST web service is defined with a resource model, shown in Figure 3.1. Resource model defined *resource definitions* of the service and their properties. The behavioral model defines the life cycle of a service using resources of the service, i.e. instances of *resource definitions*.

We use the resource structure information presented in the resource model to build REST behavioral model. Figure 4.4 shows the behavioral model of hotel room booking web service. The service can take a booking from its user and waits for the payment. If a booking is not paid within 24 hours, it is canceled by the system. When the user pays for his booking, it is processed by another web service that can either confirm or unconfirm the payment. If the third party service does not respond with the payment information within 2 hours of payment, the booking service cancels the payment. A confirmed and unpaid booking can be canceled and a canceled booking is deleted by the system after 6 months.

The atomicity of such transactions for complex scenarios and with time constraints can also be addressed using the Try-Cancel/ Confirm pattern proposed by Pardon and Pautasso [98]. Our work can take advantage of this protocol to cater the atomicity of the transactions, however, in current work we focus on how to design services with complex interactions such that the states of services are reflected in the states of the resources.

The behavioral model in Figure 4.4 has one composite state, *activeBooking*, and one simple state, *canceled* at the top level of state hierarchy. *activeBooking* is composed of composite state *notConfirmed* and simple state *confirmed*. The simple states *notpaid* and *processingpayment* are contained in *notConfirmed*.

The booking resource can be retrieved (GET) and deleted (DELETE) and the *cancel* resource can only be updated (PUT) and retrieved (GET). Similarly, one of the operations of the service is to pay a booking. This is achieved by a HTTP PUT request over a payment resource. However, a payment can only be accepted if it is connected to a room and a booking can only be paid once. Also, a booking can be canceled, but not while the payment is being processed. We need to define all these conditions in the behavioral interface of the service.

A REST interface mostly uses an HTTP protocol, so our first design requirement is that the only allowed methods that can be invoked on resources are GET, PUT, POST and DELETE and a state change can be triggered only by POST, PUT and DELETE. This provides a uniform interface. The methods are represented as: *METHOD_NAME (path,* $p_1, p_2, ..$), where *path* is the relative path of the resource on which the HTTP method is invoked, and $p_1, p_2, ...$ represent the HTTP request parameters, if any, passed with the method.

In our behavioral model, the transition triggers can only be defined as POST, PUT or DELETE operations over resources described in the resource model. The guards on the transitions and the state invariants that define the service state can be defined only using information from the resources and request parameters.

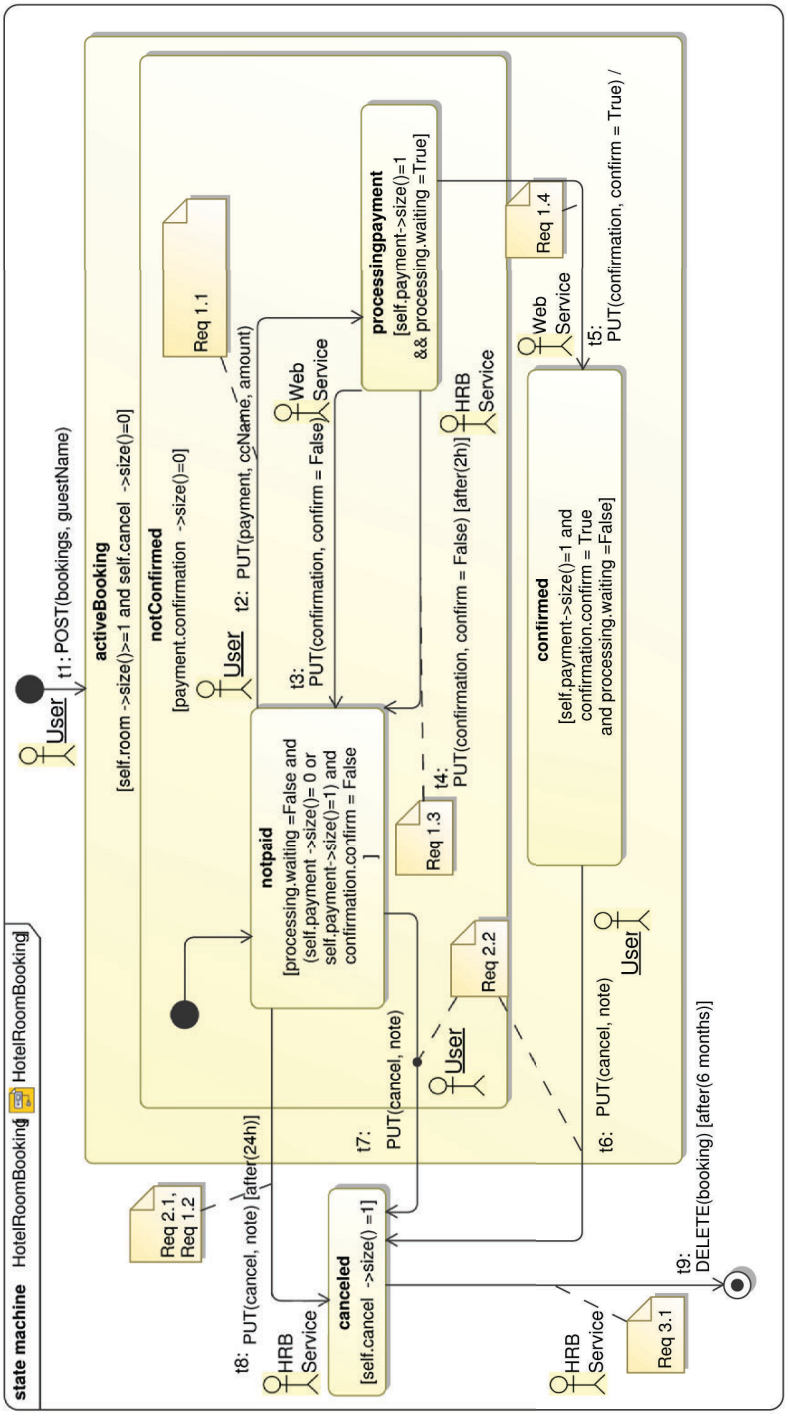


Figure 4.4: Behavioral Model for HRB RESTful Web Service

4.3.1 GET Method

The GET method retrieves the representation of the resource and it should not have side effects. For example, *GET(/bookings/{booking_id}/payment)* is HTTP GET method on the resource *payment*. Whenever a GET method is called on a resource, it gives the representation of resource as a response. In practice, the access to resources may be restricted by an authentication and access control mechanism.

4.3.2 POST Vs. PUT Method

Both POST and PUT methods can be used to create a new resource. However, these methods are invoked based on certain criteria defined for REST.

A clear difference between POST and PUT given in [111] is that the client uses POST when server is incharge of deciding what URI the new resource should have and the client uses PUT when it is incharge of deciding what URI the new resource should have. Alternatively, we can say, when a POST method is invoked on an existing URI, a new resource is created and if PUT is invoked on an existing URI, it just modifies the existing resource. A PUT creates a new resource only when it is invoked on an new URI.

POST Method: POST is generally used to create subordinate resources, i.e., resources that exist in relation to another *parent* resource [111]. and it is not idempotent. This means that invoking a POST on the same resource multiple times will always create a new resource with new address with same properties.

In our model, a POST method is used create a new resource by invoking it on a collection resource. For example, in Figure 4.4 a POST is invoked on collection resource *bookings* to create new resources. This URI of the new booking resource is returned as a part of the response.

PUT Method: PUT method is idempotent, i.e., it has the same effect if you invoke it once or more than once. If PUT is invoked multiple times on a resource, it will create a new resource the first time and would keep updating the same resource with the same address for subsequent invocations. For example, in Figure 4.4 PUT is invoked on cancel resource with PUT(/bookings/{booking_id}/cancel). It creates a new cancel resource when it is invoked the first time. If the client needs to modify it, it invokes PUT on cancel resource with the same URI with different parameter values.

4.3.3 DELETE Method

DELETE method deletes a resource. This method is also idempotent. This means that invoking a DELETE on a resource will have the same effect even if it is invoked multiple times. We delete *booking* resource by invoking DELETE on it, i.e., DELETE(/bookings/{booking_id}/). However, only a canceled booking can

be deleted by the system and if it has been canceled for more than 6 months. This is shown by allowing DELETE to trigger a transition on *booking*, only when it is in the *canceled* state and time constraints are added as a guard on the transition.

4.3.4 State Invariant

We retrieve the resource of a state by invoking GET on it. When we invoke an HTTP GET method on a resource, it returns its representation along with the HTTP response code. This response code tells whether the request went well or bad. If the HTTP response code is 200, this means that the request was successful and the invoked resource exists. Otherwise, if the response code is 404, this implies that URI could not be mapped to any resource and the invoked resource does not exist.

We use the information inferred from the response codes and representation of resources to define our service state invariants in behavioral model. We have used OCL to define state invariants in behavioral models of REST web services that are represented by UML state machine diagrams. The UML specification proposes the use of OCL to define constraints in UML models, including state invariants. OCL is well supported by many modeling tools [49, 58]. We consider OCL constructs using mainly multiplicity, attributes value and boolean operators.

Attribute Constraints

The value of the attribute is accessed in OCL by using a keyword $self$ or by using a class (resource) reference ([96],p.15), the value constraint of the attribute *Att* is written in OCL as `self.Att=Value`, meaning $\{x|(x,Value) \in Att\}$, where *Value* represents the attribute value. For example, the state invariant for the state *notConfirmed* in Figure 4.4 contains the boolean expression *payment.confirmation -> size() = 0*, where confirmation refers to attribute value *confirmation* on resource *payment* (represented by association relationship in the resource model)

Multiplicity Constraints

The multiplicity of an association is accessed by using $size()$ operation in OCL ([96],p.144). The multiplicity constraint on the association *A* in OCL is written as `self.A->size()=Value`, where *Value* is a positive integer and represents the number of allowable resources of the range resource definition of the association *A*. We can use a number of value restriction infix operators with $size()$ operation such as $=$, $>=$, $<=$, $<$ and $>$. The multiplicity constraint on an association *A* is defined as $\{x|\#\{y|(x,y) \in A\} OP\ Value\}$, where *OP* is the infix operator and *Value* is a positive integer. For example, consider the state invariant for the state *activeBooking* in Figure 4.4. *self.room -> size() >= 1* checks the response code for the HTTP GET methods on the resource *room*. It evaluates to true if response code of the invocation, i.e., GET on *room* for a particular booking ID ({booking_id}) is

200. Similarly, *self.cancel -> size() = 0* is true if response code of the invocation, i.e., GET on *cancel* for a particular booking ID({booking_id}), is 404. For the hotel room booking service to be in state *cancel*, the state invariant of this state should be true.

Boolean Operators

The constraints in a state invariant are written in form of a boolean expression, and joined by using the boolean operators, such as "*and*" and "*or*" ([96],p.144). For example, consider the state invariant for the state *activeBooking* in Figure 4.4. The boolean expressions *self.room -> size() = 1* and *self.cancel -> size() = 0* are combined with boolean operator *and*.

We must note that to evaluate the state invariant of a substate, its state invariant should be conjuncted with state invariant of all the super states that contain it.

4.3.5 More on Connectedness

A stateful service may impose a certain sequence of method invocations that must be respected in order to achieve service goals. This sequence of method invocation is evident by looking at the behavioral model.

Also, since the method invocations are dependent on states of the service that cannot be maintained via sessions in a stateless protocol, the service representations should contain a list of links that can be further invoked providing *connectivity* feature to REST interface. This feature allows the client to follow the right sequence of method calls. We require that this information should be formulated from the behavioral model.

When a POST or a PUT method is invoked on a resource, it returns the resource representation along with the status code. In addition to resource attributes, the resource representation also contains list of links that can be invoked further. When an HTTP method invokes a transition in behavioral model, we observe the outgoing transitions from its target state. The trigger of these outgoing transitions become part of the resource representation and returned along with with HTTP response codes to the client. This information allows the service client to follow a trail of method invocations and informs the client about the allowed HTTP methods that can be invoked on a resource. Thus, the service carries forward its operation in a stateful manner treating hypermedia as the engine of service states.

4.4 Synchronous and Asynchronous Web Services

Interaction between web services can be either synchronous or asynchronous. This interaction is distinguished in the manner request and response are handled. When a client invokes a synchronous services, it suspends further processing until it gets a response from the service. On the other hand, when a client invokes

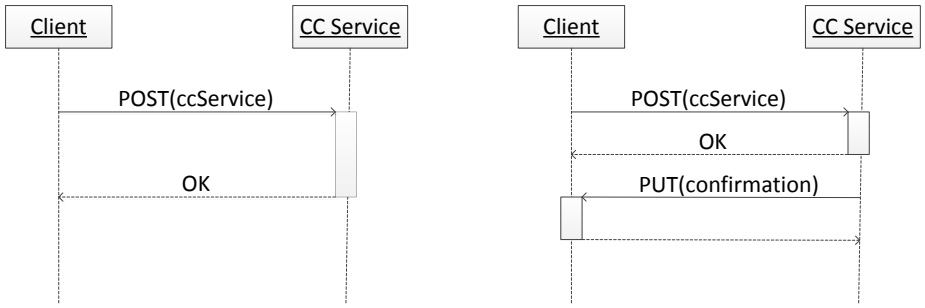


Figure 4.5: (Left) Interaction with Synchronous CC Service. (Right) Interaction with Asynchronous CC Service

an asynchronous service it does not wait for the response and continues with its processing. The asynchronous service can respond later in time. The client receives this response and continues with its processing.

We have modeled the scenario for asynchronous third party service in Figure 4.4 by creating a *processing* state in our state machine. A third party credit card payment service is invoked when a PUT is invoked on the payment resource. This would invoke the credit card service as an internal action that is not shown here since it is not part of interface operations of web service. The credit card payment service is an asynchronous service so it may take a long time to process the credit card and confirm the payment back to the client. Thus, the system goes into a processing state for that particular booking with booking ID {booking_Id} and resumes processing of other transactions. When the confirmation response is received from the third party service, the processing for this booking is resumed.

It may be worth pointing out that the agent POSTing the payment (the client) must also be able to act as a server in order to receive a PUT payment confirmation. As an alternative, the credit card service might return 202 (Accepted) response with location. This would require the client to poll for confirmation.

4.5 Authorization and Actors

In a secure web, the requests must be authenticated to ensure that the request is coming from the right party and if the consumer of the service is authorized to access the privileged resource. In this context, our behavioral model uses the notion of actor to specify which party invokes a certain method.

An actor of the service is the participant that invokes different methods on different resources of the service resulting in different service states. In Figure 4.4, the involved parties are annotated as actors on the transitions along with the methods they trigger, guards and request parameters. The *user* can invoke POST

Table 4.1: Requirements Table of Hotel Room Booking REST Web Service

Req	Sub-Requirements
1- Payment	1.1 - A booking should be paid by the user 1.2 - If a booking is not paid within 24 hours, then it is automatically canceled by the system (HRB Service) 1.3 - If payment processing does not confirm payment within 2 hours, then it is automatically canceled by the system. 1.4 - If the payment is successful then the booking must be confirmed.
2- Cancel	2.1 - A booking is canceled by the system (HRB service) if it is not paid for 24 hours 2.2 - A booking can be canceled by the user only if it is not waiting for payment processing, i.e., a booking can be canceled only if it is unpaid or confirmed.
3- Delete	3.1 - A canceled booking is deleted automatically by the system (HRB service) after 6 months

and DELETE on *bookings* and PUT on *payment* resource, where as the partner *web service* invokes the *confirmation* resource with True or False information. A booking can be canceled by the *user* and the system, i.e., *HRB Service*.

The service developer can use the information of actors provided in the behavioral model to implement the access rights on resources during implementation. When the service user makes an HTTP request on a privileged resource, it provides its credentials in the authorization header. If the credentials are wrong, consumer is denied access to the resource.

4.6 Domain-Specific Requirements

Requirements can be decomposed in different categories like functional, architectural, temporal, data etc and are generally domain specific since they are inferred from the specification document. We infer functional and temporal requirements from the specification document into a table and number them. These requirements are attached to the behavioral model as comments on the transitions or states. When a state or transition with the requirement annotation is traversed, it indicates which service goal is met.

Table 4.1 shows the requirements for hotel room booking REST web service that are added as comments on Figure 4.4. All these requirements must be met by the service implementation in order to satisfy all service goals.

4.7 Time Constraints

Service specifications may impose certain time restrictions on service implementations to ensure timely delivery of service operations. This allows the service not to wait uselessly for a service that does not respond. We require that these time constraints should be captured at the design time in order to carry them forward in all the other phases of service development. This makes design models amenable for verification of their timed behavior before implementing them.

In our behavioral model, the time requirements are added using UML time events for state machines. In our example of hotel room booking service, we require that a booking is canceled if it is not paid for 24 hours. Similarly, if the payment processing service does not confirm the payment within 2 hours, it is marked as unpaid and a canceled booking is automatically deleted by the system after 6 months. We model these properties in behavioral model in Figure 4.4 by adding *after* time event as guards on the respective transitions.

4.8 Stateless State Machines?

REST web services offer a stateless interface. Using a state machines to model a stateless interface may seem like an oxymoron. In the context of a RESTful service, statelessness is interpreted as the absence of hidden information kept by the service between different service requests. In that sense, a RESTful web service should exhibit a stateless protocol. Also, there is no sense of session or sequence of request in a RESTful service.

On the other hand, state machines have a notion of active state configuration, that is, what states are active at a certain point of time. If an implementation of an interface described using a state machine would have to keep the active state configuration between different requests, then this would break the statelessness requirement of the RESTful service.

However, our approach does not actually require that a service implementation keeps any additional protocol state. In our approach a state is active if its invariant evaluates to true and the invariants are defined using addressable resources. Therefore an implementation of a service can determine the active state configuration by querying the service state. There is no need to keep any additional protocol state.

4.9 Well-formedness Rules for Behavioral Model

Our behavioral model is represented by UML protocol state machine with certain design restrictions. These restrictions must be taken into consideration by the developer when constructing models for REST web service. A list of well-formedness rules for behavioral model, inferred from the discussions above, is given below:

- Every state should have a state invariant.

- Method calls should be either PUT, POST and DELETE.
- The state invariants of each state should be mutually exclusive .
- Each requirement should be attached to a transition.

4.10 Conclusion

A behavioral model captures the dynamic structure of a REST web service using UML protocol state machine. In this chapter, we have shown how the different properties of REST behavioral interface are modeled. We show how different REST features are modeled by introducing different design constraints on protocol state machine. We covered method calls, connectivity, synchronous and asynchronous services, authorization, timed behavior along with other REST features. Our approach advocates creation of REST interfaces if the well formedness rules of service design models are followed by the service developer. We also presented our approach to generate behavioral interfaces from UML protocol state machine. In the next chapter we discuss how the behavioral interfaces for REST web services are generated from behavioral models.

Chapter 5

From Service Design Models to a REST Interface

Service designs models contain behavioral information such that every transition provides manyfold information. This information contributes to the creation of web service interfaces that exhibit RESTFul behavior. In this chapter, we show how the service design models lead to a REST interface.

In the previous chapter, we had studied how the method contract information can be generated from UML protocol state machines and asserted into code. Our understanding of behavioral model of REST web service interface is much clearer now so, in section 5.1, we explain how the contract generation approach is applied on behavioral model to create behavioral interfaces of a REST web service.

Web Application Description Language (WADL) [121] provide service descriptions for REST web services. In section 5.2 we discuss the generation of behavioral WADL service descriptions.

Each HTTP request on a service is followed by an HTTP response that contains resource representation. In section, 5.3 we show the pairs of HTTP requests and their responses containing information inferred from the service design model. Section 5.4 concludes the chapter.

5.1 Method Pre- and Post Conditions

Method contracts specify conditions under which a method should be invoked and the expected result of method invocation. This constraint the user to invoke the service under the right conditions and the developer to rightly implement the expected functionality.

However, determining what is the active state configuration of the interface state machine every time that a service implementation has to fulfill a request may be a slow task in the case of complex interfaces with many states. However, in practice it is not necessary to explore all states in the state machine but only the

source states of the transitions that can be triggered based on the current request. We have discussed in Section 4.3 about how we can do that by computing the precondition (and postcondition) of each method request. We are now interested in discussing detail how this behavioral interface is computed and implemented for REST web services.

When the state invariant of a state is true, we say that this service state is active otherwise false. In doing so for a REST service we take advantage of the fact that the service states can be defined as predicates over resources. This means that GET methods are invoked on different resources of the service and combined as a boolean expression to form a state invariant. HTTP GET methods are free of any side-effects.

Our aim is to generate pre conditions and post conditions of side-effect methods. The precondition of a method tells under what conditions a method can be triggered. We say that the precondition of a method m is satisfied when the state invariants of all the source states of transition t are true along with its guard condition.

In a similar manner, if a method m triggers a transition t in a behavioral model, then its post-condition is satisfied when the state invariants of all the target states of transition t are true along with the postcondition on the transition t.

For the details and formal definitions of generating preconditions and postconditions for different elements in a UML protocol state machine of a class readers are referred to section 4.2.

We apply the same definition of contract generation on our behavioral model, reproduced in Figure 5.1 for different cases. These cases are simple transitions, join and fork transitions, high level transitions, conflicting transitions and cases of priority transition and non-deterministic behaviors.

5.1.1 HTTP Method Pre-Condition

The precondition of a method is given by creating a boolean expression of state invariants of all the source states of transitions, to which that method is a trigger, and its guard conditions. We express GET method invocations on resources as OK(r) and NOT_FOUND(r) functions for a concise representation in the listing. Here, OK(r) represents *self.r -> size() = 1* and True value of an attribute. NOT_FOUND(r) represents *self.r -> size() = 0* and False value of an attribute. To recall, *self.r -> size() = 1* represents a GET method call on resource r with response code of 200 and *self.r -> size() = 0* represents a GET method invocation on resource r with response code of 404.

According to Definition 8 (in section 4.3.3), a precondition for a method m is defined as:

$$precondition(m, \sigma) = \exists t \in T : enabled(t, m, \sigma)$$

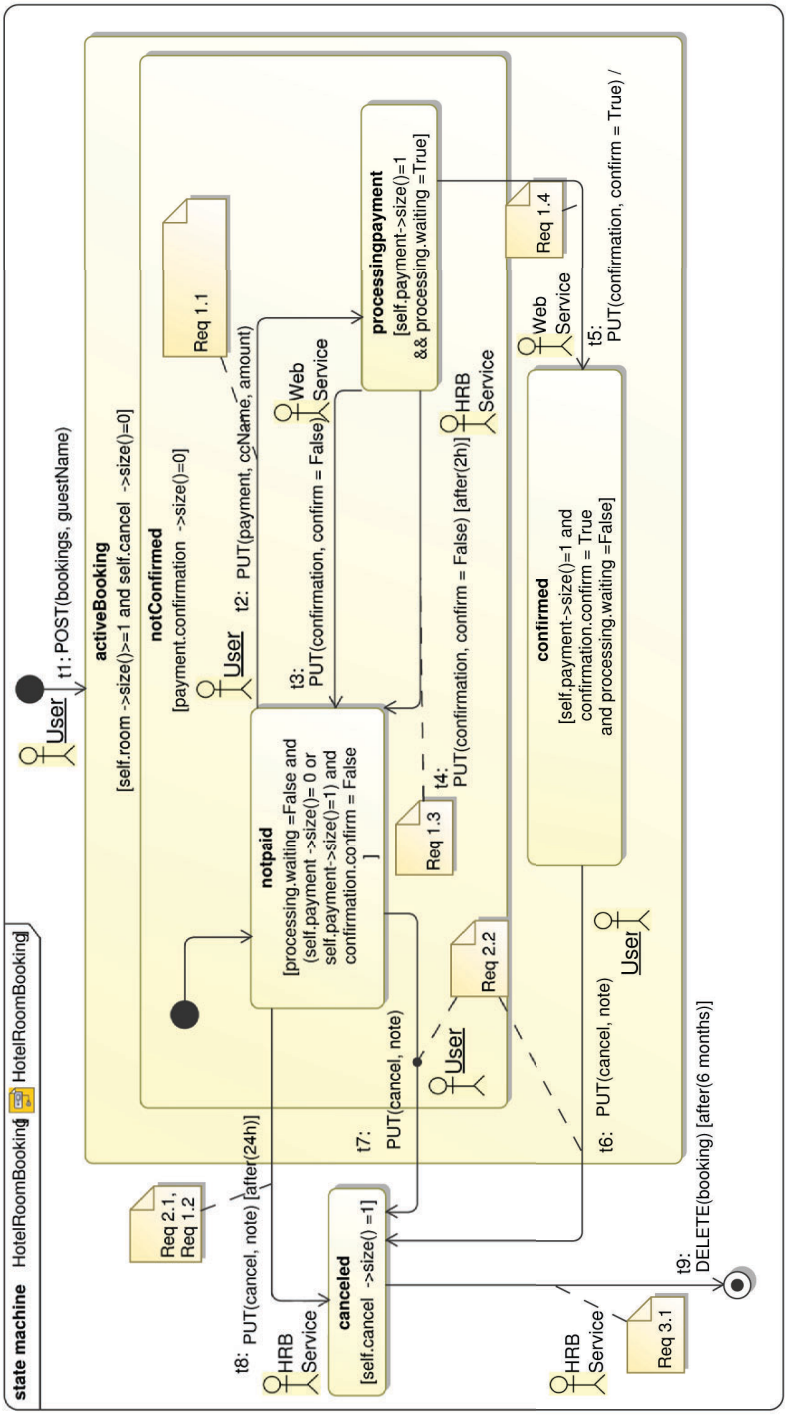


Figure 5.1: Behavioral Model for HRB RESTful Web Service

We apply this definition to calculate precondition for our HTTP methods in behavioral model. The precondition for PUT on cancel, in Figure 5.1, is given as:

$precondition(put\_cancel, \sigma) = enabled(t7, put\_cancel, \sigma) \vee$
$enabled(t8, put\_cancel, \sigma) \vee enabled(t6, put\_cancel, \sigma)$

By replacing Definition 2 (from section 4.3.3) of *enabled* for each transition, we obtain the following expansions for transitions *t7*, *t8* and *t6*:

$enabled(t7, put\_cancel, \sigma) = (OK(room)$ *and* $NOT\_FOUND(cancel))$ *and*
$(NOT\_FOUND(processing.waiting)$ *and* $(NOT\_FOUND(payment)$ *or*
$OK(payment))$ *and* $NOT\_FOUND(confirmation.confirm))$

$enabled(t8, put\_cancel, \sigma) = (OK(room)$ *and* $NOT\_FOUND(cancel))$ *and*
$(NOT\_FOUND(processing.waiting)$ *and* $(NOT\_FOUND(payment)$ *or*
$OK(payment))$ *and* $NOT\_FOUND(confirmation.confirm))$ *and* $time > 24$

$enabled(t6, put\_cancel, \sigma) = (OK(payment)$ *and*
$NOT\_FOUND(processing.waiting)$ *and* $OK(confirmation.confirm))$

After replacing definitions of *enabled*, for all the transitions, into precondition equation, we get the precondition for PUT on cancel as shown in Listing 5.1.

5.1.2 HTTP Method Post-Condition

The postcondition of a transition will be evaluated only if the precondition for that transition is true. We define as *pre_OK(r)* the function that gives boolean value of *self.r -> size() = 1* before invoking the trigger method. Similarly, *pre_confirmation.confirm* and *pre_NOT_FOUND(r)* give the representation of attribute *confirm* and boolean value of *self.r -> size() = 0* before invoking the trigger method, respectively.

The postcondition for a method m is given in Definition 9 in section 4.3.3. We refine this definition for PUT on cancel as follows:

$postcondition(put\_cancel, \sigma, \sigma') = enabled(t7, put\_cancel, \sigma) \implies$
$effect(t7, \sigma')) \wedge enabled(t8, put\_cancel, \sigma) \implies effect(t8, \sigma')) \wedge$
$enabled(t6, put\_cancel, \sigma) \implies effect(t6, \sigma'))$

The effect of all the transitions are same since they all go to the same target state. Hence:

$effect(t6, \sigma') = OK(booking)\ and\ OK(cancel)$
$effect(t7, \sigma') = OK(booking)\ and\ OK(cancel)$
$effect(t8, \sigma') = OK(booking)\ and\ OK(cancel)$

Listing 5.1 also shows the postcondition for PUT on *cancel* resource, generated from Figure 5.1 , after replacing the definitions of enabled and effect in the equation of postcondition.

Listing 5.1: High Level Contract for PUT on cancel

```
PATH
 booking: bookings/{booking_id}/
 room: bookings/{booking_id}/room/
 confirmation: bookings/{booking_id}/payment/confirmation/
 cancel: bookings/{booking_id}/cancel/

PUT bookings/{booking_id}/cancel/
 precondition
  ((OK(room) and NOT_FOUND(cancel)) and
  (NOT_FOUND(processing.waiting) and
  (NOT_FOUND(payment) or OK (payment)) and NOT_FOUND(confirmation.
      confirm))
  or ((NOT_FOUND(processing.waiting) and (NOT_FOUND(payment) or OK (
      payment))
  and NOT_FOUND(confirmation.confirm) and time > 24)))
  or (OK(payment) and NOT_FOUND(processing.waiting) and OK(
      confirmation.confirm) )))

 postcondition
  ((( pre_OK(room) and pre_NOT_FOUND(cancel)) and
  (pre_NOT_FOUND(processing.waiting) and (pre_NOT_FOUND(payment) or
      pre_OK (payment)) and pre_NOT_FOUND(confirmation.confirm))
  or ((pre_NOT_FOUND(processing.waiting) and (pre_NOT_FOUND(payment)
      or pre_OK (payment)) and pre_NOT_FOUND(confirmation.confirm)
      and time > 24))) ==> OK (booking) and OK(cancel)) and
  (pre_OK(payment) and pre_NOT_FOUND(processing.waiting) and pre_OK(
      confirmation.confirm) )  ==>  OK (booking) && OK(cancel)))
```

Previously, we had shown how the contracts can be asserted as JML specifications in a Java class. We can assert JML specifications in a similar manner in a java based web service. For web services implemented in other programming languages such as python, this is just a simple exercise of mapping pre and post conditions in that language. In chapter 9, we have shown how the method contracts are asserted in an automated manner in python based web services developed using Django web framework.

5.2 Generation of Behavioral WADL Service Descriptions

Web Application Description Language (WADL) is used for publishing RESTful web service interfaces and provides a machine-processable description of the

interface [121]. Currently, RESTful architectural style and WADL are being widely adopted in the web and have numerous users, including enterprizes such as Google, Yahoo, Amazon and Flicker.

The information about the interface of a web service in WADL is syntactic and does not say anything about its semantics, i.e., how a service should be invoked and behave.

We extend WADL to include information about the behavior of the methods in a service. Our objective is to generate this information automatically from the resource and behavioral models described above.

WADL defines the operations that can be invoked on an interface and describes the input and output parameters for each operation. It defines the resources that a service contains and methods that can be called on them. Each method has two attributes *name* and *id*, where *name* is the name of the HTTP method and *id* is the ID of the method that is associated with the HTTP method.

Representing information in the resource model as part of a WADL service description is a rather straight forward task. However, the behavioral model does not map directly to a WADL description since the behavioral model allows different transitions to be triggered by the same method. In our example, a cancel request can be invoked when the service is in different states. That is the information about when a method can be invoked (precondition) and what is its expected result (postcondition) needs to be computed from the different states in the behavioral model as explained in section 5.1.

5.2.1 Inserting Pre- and Post Conditions into WADL Service Descriptions

We refine the high-level contracts presented in section 5.1 with details of the relative navigation paths, the invoked HTTP methods and the expected response codes. These refined contracts are asserted into WADL interface. The function *pre_OK(r)* is mapped to a *pre_GET* function and its response code is compared to 200. The *pre_GET(r)* function gives the stored results of invoking a GET method on resource *r* before invoking the method. In similar manner, a *pre_NOT_FOUND* is mapped to a *pre_GET* function and its response code is compared to 404.

To support the behavioral information in interface descriptions, we extended the XML schema of WADL with two elements *precondition* and *postcondition*, with an attribute *id* for each of these elements. These tags, i.e., $<$ *precondition* $>$ and $<$ *postcondition* $>$ are asserted above and under the method tag, respectively. Similarly, we have added an element *clock* that is of type *xs:time* to represent time.

An excerpt of a behavioral RESTful interface is shown below for method PUT on *cancel* resource.

```
<resources base = "http://www.example.com/bookings">
 ...
 <resource path = "{booking_id}">
```

```
...
 <resource path = "cancel">
  <precondition id = "pre_put_cancel" >
   (GET(/bookings/{bid}/room/) == status(200) &&
    GET(/bookings/{bid}/cancel/) == status(404))
   &&
   ((GET(/bookings/{bid}/payment/processing == status
       (404)) &&
   (GET(/bookings/{bid}/payment/) == status(200) || (
      GET(/bookings/{bid}/payment/) == status(404)))
   && GET(/bookings/{bid}/payment/confirmation) ==
       status(404))
   ||
   (GET(/bookings/{bid}/payment/processing == status
       (404)) &&
   (GET(/bookings/{bid}/payment/) == status(200) || (
      GET(/bookings/{bid}/payment/) == status(404)))
   && GET(/bookings/{bid}/payment/confirmation) ==
       status(404) and clock > 24))
   ||
   (GET(/bookings/{bid}/payment/) == status(200) && GET
       (/bookings/{bid}/payment/confirmation) == status
       (200) && GET(/bookings/{bid}/payment/processing)
       == status(404))
  </precondition>
  <method name = "PUT" id = "cancel"> </method>
  <postcondition id ="post_put_cancel" >
   (pre_GET(/bookings/{bid}/room/) == status(200) &&
    pre_GET(/bookings/{bid}/cancel/) == status(404))
   &&
   ((pre_GET(/bookings/{bid}/payment/processing ==
       status(404)) &&
   (pre_GET(/bookings/{bid}/payment/) == status(200) ||
        (pre_GET(/bookings/{bid}/payment/) == status
       (404)))
   && pre_GET(/bookings/{bid}/payment/confirmation) ==
       status(404))
   ||
   (pre_GET(/bookings/{bid}/payment/processing ==
       status(404)) &&
   (pre_GET(/bookings/{bid}/payment/) == status(200) ||
        (pre_GET(/bookings/{bid}/payment/) == status
       (404)))
   && pre_GET(/bookings/{bid}/payment/confirmation) ==
       status(404) and clock > 24)) ==> (GET(/bookings/{
       bid}/) == status(200) && GET(/booking/{bid}/
       cancel)== 200))
    &&
```

```
      ( (pre_GET(/bookings/{bid}/payment/) == status(200)
          && pre_GET(/bookings/{bid}/payment/confirmation
          ) == status(200) && pre_GET(/bookings/{bid}/
          payment/processing) == status(404)) ==>
      (GET(/bookings/{bid}/) == status(200) && GET(/
          bookings/{bid}/cancel)== 200))
   </postcondition>
  </resource>
   ....
  </resource>
 </resources>
```

5.3 HTTP Requests and Responses

HTTP requests are invoked on different resource of REST web services. When a user invokes an HTTP method on a service, it is returned an HTTP response containing status code and other information (if any). When an allowed HTTP request is made on a resource, its HTTP response is sent with entity headers, status code and the representation of the resource in that state (if content exists). The representation of a resource consists of attributes of the resource and hyperlinks that can be navigated further.

In Figure 5.1, we have modeled the different HTTP methods that can be invoked on our example service. Table 5.1 shows the possible HTTP requests and their expected responses on the hotel room booking REST web service. The request parameters with HTTP requests map to the parameters of HTTP methods in Figure 5.1. HTTP responses contain the expected HTTP response code and the representation of the invoked resource including the attributes that are defined in its resource model.

The link array has the relative paths of the transitions that can be taken from the target state to which the invoked HTTP request is a trigger. These links are created by following the outgoing transitions from the target state of the transition in question. We represent link element in JSON by two attributes `href` and `rel` as defined in [14]. `href` has the URI and `rel` gives name of the resource to which the hyperlink points. It is considered a good practice to provide HTML documentation at the URI containing information about purpose of the link and its valid HTTP methods alongwith the expected media types [14].

As an example, lets take a POST request on *bookings* resource in Table 5.1. The POST request is invoked on *bookings* resource with request parameter *guestName*. Its HTTP response contains response code 201 (created), JSON representation of the created resource and the links information for *payment* and *cancel* resources. This array of links contains links to resources that can be navigated further to get the desired functionality from the stateful web service. The array does not contain

links to confirmation resource since invoking a confirmation resource at this point of service cycle will provide no results.

5.3.1 HTTP Authentication

The information of actors available in the behavioral model facilitates the developer to implement the access rights on resources and helps service users to understand and write correct authorization headers. Different authentication mechanisms can be implemented to control access to resources [5]. In case Basic authentication mechanism is implemented, client sends the user name and password to the server in authorization header. The authentication information is in base-64 encoding. It should only be used with HTTPS, as the password can be easily captured and reused over HTTP.

The authorization header is constructed by first combining username and password into a string "username:password" and then encoded in based64. A typical authorization header in Basic authentication is shown below:

```
GET /bookings/1/cancel/ HTTP/1.1
Host:http://www.example.com/bookings/
Authorization: Basic aHR0cHdhdGNoOmY=
```

In case an anonymous requests for a protected resource, HTTP can enforce basic authentication by rejecting the request with a 401 (Access Denied) status code.

```
HTTP/1.1 401 Access Denied
WWW-Authenticate: Basic realm="Hotel Room Booking Server"
Content-Length: 0
```

5.4 Conclusion

In our design approach, we have presented resource and behavioral models that we claim can create behavioral interface of a REST web service. A REST interface should exhibit these four attributes: *addressability, connectivity, statelessness* and *uniform interface*. Our service design models lead to web services that exhibit these attributes and make our interfaces REST compliant.

We constrain our resource model to be a connected graph such that no resource is isolated. Each resource can be addressed independently using the navigation directions of associations and their role names. The role names give the relative navigation path between the resources. Thus, each resource has a URI address providing *addressability* feature to our resource model. Our resource model does not contain any method information. The methods that can be invoked on a resource are inferred from the behavioral model. We restrict the behavioral model so that transitions can only be triggered using the standard HTTP methods, providing the *uniform interface* feature.

Table 5.1: HTTP Request and Response pairs for Hotel Booking

Request	Response
POST /bookings/ HTTP/1.1 Host: www.example.org Content-Type: application/json {‘guestName’: ‘Mary’ }	Http/1.1 201 Created Location: http://www.example.org/ bookings/021 Content-Type: application/json {‘bid’: ’021’, ‘bdate’: ‘20-11-2009’, , ‘guestName’: ‘Mary’, ‘link’:[{ ‘rel’: ‘payment’, ‘href’: ‘/bookings/021/payment/’}, {‘rel’: ‘cancel’, ‘href’: ‘/bookings/021/cancel/’}]}
PUT /bookings/021/ payment/ HTTP/1.1 Host: www.ex... Content-Type: application/json { ‘ccName’: ‘Richard’, ‘amount’: ‘142’}	Http/1.1 200 OK Location: http://www.example.org/ bookings/021/payment/ Content-Type: application/json {‘pid’: ‘10’, ‘bookingid’: ‘021’ , ‘amount’: ‘350’,‘pDate’: ‘11-10-2010’, ‘ccName’: ‘Richard’, ‘link’:[{‘rel’: ‘confirmation’, ‘href’: ‘/bookings/021/payment/confirmation/’}, {‘rel’: ‘cancel’, ‘href’: ‘/bookings/021/cancel/’}]}
PUT /bookings/021/payment/ confirmation/ HTTP/1.1 Host: www.example.org Content-Type: application/json {‘confirm’: ‘True’ }	Http/1.1 200 OK Location: http://www.ex.../ bookings/021/payment/confirmation/ Content-Type: application/json {‘confirm’: ‘True’, ‘link’:[{‘rel’: ‘cancel’, ‘href’: ‘/bookings/021/payment/cancel}]}
PUT /bookings/021/cancel/ HTTP/1.1 Host: www.example.org Content-Type: application/json {‘note’: ‘not traveling’ }	Http/1.1 200 OK Location: http://www.ex...//bookings/021 /cancel/ Content-Type: application/json {‘cdate’: ‘11-11-2010’, ‘note’: ‘not traveling’} ‘link’:[{‘rel’: ‘booking’, ‘href’: ‘/bookings/021/’},
DELETE /bookings/021/ HTTP/1.1 Host: www.example.org	Http/1.1 204 No Content

In addition, we have created stateful service using stateless service interface thanks to the fact that the service states are defined using state invariants defined in terms of exposed resources. This information is captured in the behavioral model of the REST interface, providing the *statelessness* feature in our behavioral model.

The behavioral model also specifies the transitions that can be taken from a give service state. This is inferred from the trigger methods on the outgoing transitions of a service state and specified as links in the *representation* of resource in response to the HTTP request. This leads consumers through trail of resources resulting in service state transitions providing *connectivity* feature to our REST interfaces.

We also provide clear mapping to HTTP requests and responses for the REST behavioral interface. Transitions are labeled with request parameters, that are part of HTTP request, and also with the service actor who is allowed to invoke the method. This information can then be used to authorize users by authenticating the requests on the privileged resources and generate appropriate HTTP responses.

The models can be also used to generate a contract in the form of preconditions and postconditions of interface methods. These contracts can be included in a WADL interface specification.

The behavioral RESTful interfaces have many applications. They can serve as a documentation for existing services or as a blue print to develop new ones. The models can be used to generate implementation stubs in commonly used web frameworks like Django and Ruby on Rails. They can also be used to monitor the interaction between a service and its clients and report if any of the parties breaches the interface contract. The generation of test cases from interface contracts is also a promising application of behavioral interfaces that we address in the later part of this thesis.

Chapter 6

Consistency Analysis of REST Web Service Interface

The design phase of a software development lifecycle is crucial in the development of a dependable software, since the design models developed in this phase are carried forward to all the other phases. It is therefore important that these design models are constructed correctly. The design models are created from different viewpoints to capture different features of the system under development, since all the features are difficult to capture in a single model and can make a single model complex. Capturing the system under development in different models from different viewpoints gives a better and simpler understanding of the system, but raises the issue of models inconsistency. Models can become inconsistent, if they define the same system but have contradicting specifications in different models or have specifications that cannot be satisfied resulting in implementation that can have unwanted results. The design models thus need to be analyzed for their inconsistency. The need for consistency analysis of models can rise even if the models themselves have no errors. Designers may specify certain requirements in different models that contradict each other and thus, cannot exist together leading to inconsistent diagrams. These mistakes can lead to implementations that do not provide correct functionality as expected from them.

As the software shifts from software as a product to software as a service, the need for consistency analysis of design models rises even further since web services are offered from remote locations that consumers use via Internet using standard Internet protocols. They can be expensive in terms of bandwidth and other costs. It is, therefore, important to deliver services that are dependable and do not contain unintended design mistakes to avoid undesired results.

Designing and publishing a REST web service interface with stateful behavior may involve many resources and different service states that are dependent on these resources. The service state represents a certain condition that is true when the state is active. The condition can be defined explicitly in the form of state

invariant. The state invariants for stateful REST web services are defined as predicates over resources. If these state invariants are inconsistent, they can lead to implementations with unwanted results. The inconsistent state invariants are design errors and, in order to reduce development costs and time, they must be detected and corrected as early in the software development process as possible.

In this chapter, we discuss a consistency checking approach for the service design models to detect such inconsistencies based on the use of the automatic reasoning tools developed initially in the context of the semantic web. We first translate the resource and behavioral diagrams with state invariants to the Web Ontology Language version 2 for Description Logic (OWL 2 DL) [93], and then use an OWL 2 DL reasoning tool [118, 108, 124] to determine the consistency of the design models represented as UML diagrams. Our consistency checking approach analyzes the REST design models to detect inconsistent behavior and as such advises the developer to correct the detected design mistakes and create consistent behavioral REST web service interfaces. A behavioral interface is said to be consistent if it does not contain any contradicting specifications and there exists a service that can satisfy it.

The motivation for consistency analysis is motivated in Section 6.1. The problem of determining the consistency of service design models is defined in Section 6.2 along with an overview of reasoning tool chain. An overview of description logic and OWL2 functional syntax is given in Section 6.3. Section 6.4 presents the translation from resource and behavioral models to OWL2 DL which is then analyzed in Section 6.5 using OWL 2 reasoning tool. The related work is presented in Section 6.6 and Section 6.7 concludes the chapter.

6.1 REST Design Models and their inconsistencies

The resource and behavioral models of REST web service interface are represented using UML class and protocol state machine diagrams with design constraints. In this chapter, we use the same example of hotel room booking service that we have used earlier, with small refinements. Its resource model and behavioral models are shown in Figure 6.1 and Figure 6.2, respectively. The service takes payment from the customer and books a room in the hotel. It reserves a room for the customer and uses a third party payment service for confirmation. The service can be canceled when it is not processing payment and can be deleted only if it is canceled.The example is simple to understand and helps in demonstrating complex service states.

However, we do not cater information about timing constraints, authorized actors and domain specific requirements in our consistency approach. We mainly focus on analyzing the consistency of service design models with REST constraints.

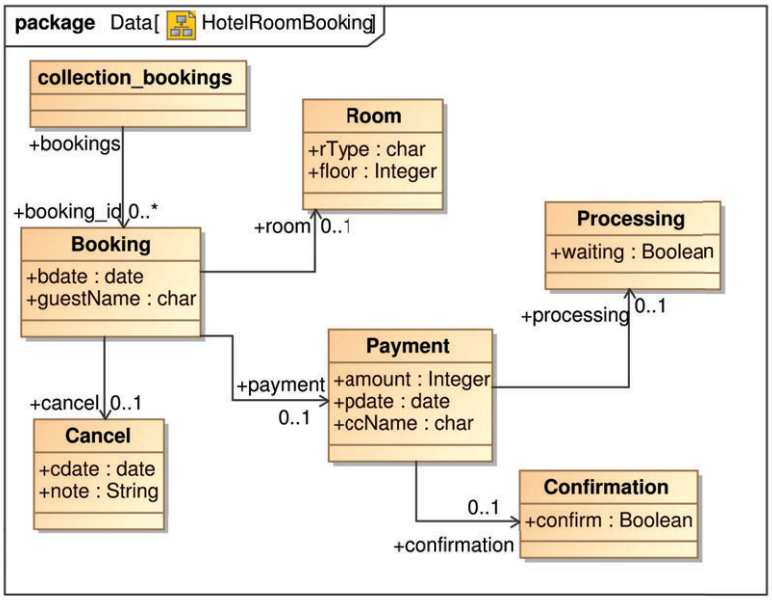


Figure 6.1: Resource Model for HRB RESTful Web Service

6.1.1 Linking Resource and Behavioral Models and Inconsistency Problems

Each service state has a state invariant. We define invariants of states as predicates over resources defined in the resource model and that can have either true or a false value. For a state to be active, its state invariant should be true, otherwise it should be false. When the client makes a service request, it is mapped to a transition in the behavioral model that has that method as a trigger. The transition is fired from a source state to a target state. If the state invariant of source state is inconsistent, a service can never exist in this state and it would be impossible for the implementation of the interface to decide which transition to take as a result of a service request.

We consider the state invariants which let the behavioral model behave against the UML superstructure specifications for statechart diagrams [125] as inconsistent state invariants, and they may cause whole system become unsatisfiable or inconsistent. The examples of inconsistent state invariants are as follows:

Inconsistent State Invariant Example 1: According to the UML superstructure specification, invariants of non-orthogonal states must be mutually exclusive ([125], p.564). For example in Figure 6.2, the state invariant of state *processingpayment* is $self\ payment\ > size() = 1\ and\ payment\ processing\ > size() = 1$. If we change its invariant to have $self\ payment\ > size() = 1\ and\ payment\ processing\ > size() = 0$, then it could conflict with the state invariant of *not paid*, i.e., both

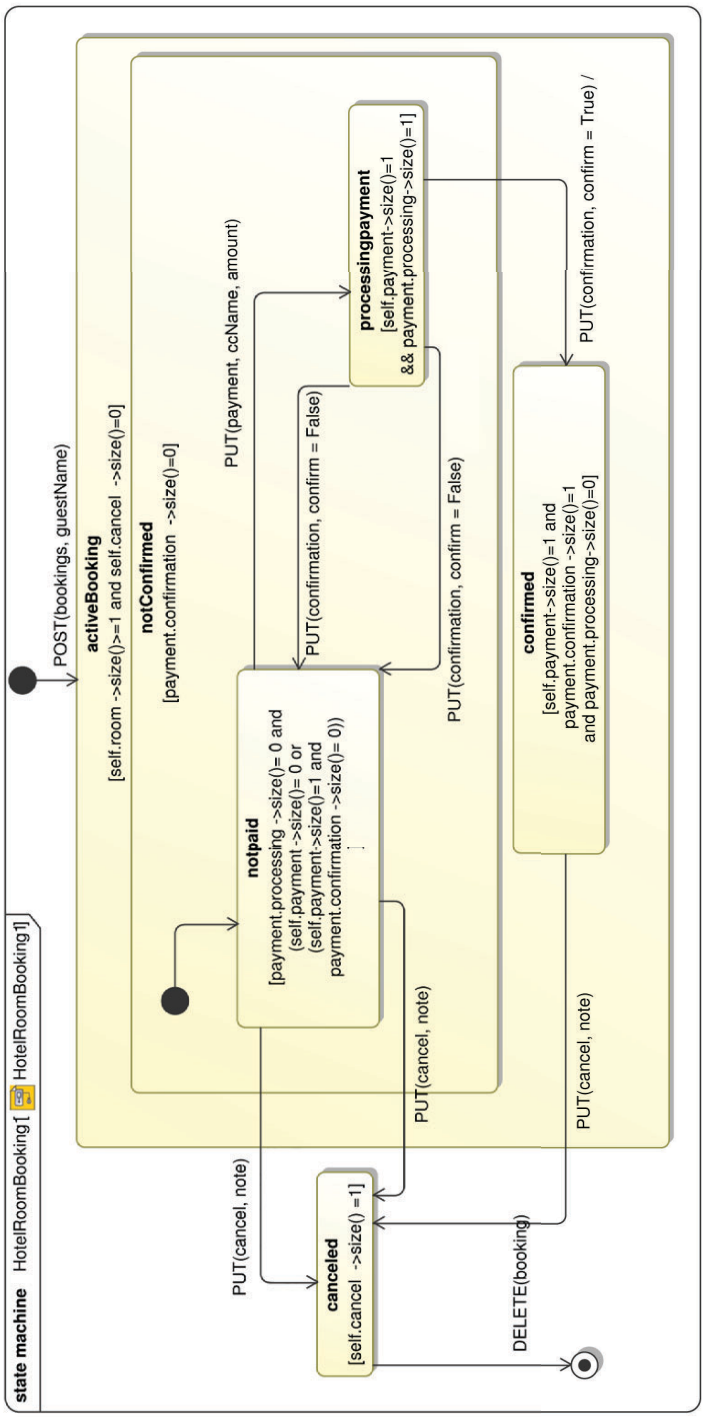


Figure 6.2: Behavioral Model for HRB RESTful Web Service

the states can be true at the same time. This would make states *processingpayment*, *notpaid* and *activeBooking* inconsistent. In this case, if PUT is invoked on *payment* resource, the implementation would not know which transition to take.

Inconsistent State Invariant Example 2: According to the UML superstructure specification, whenever a state is active, all its superstates are active ([125], p.565), means all invariants of an active state and its superstates directly or transitively are true. For example in Figure 6.2, if the state *processingpayment* is active then its superstate *activeBooking* should be also active. If we introduce an error by adding the condition `self.cancel->size()=1` in the invariant of the state *processingpayment*, this means that a canceled booking can also be processed for payment. The introduced error causes the contradiction between the invariants of the state *processingpayment* and its superstate *activeBooking*, and violates the UML superstructure specification of the behavioral diagram, and consequently makes the invariant of the states *processingpayment*, *activeBooking* and *canceled* inconsistent. Such inconsistency problems can lead to service implementations with undesirable behavior.

6.2 Consistency Analysis

In this section we define the problem of determining the consistency of our REST web service design models. Our view of model consistency is inspired by the work of Broy et al. [36]. This work considers the semantics of a UML diagram as their denotation in terms of a so-called system model and defines a set of diagrams as consistent when the intersection of their semantic interpretation is nonempty.

In our work, we assume that there is a nonempty set $\Delta^{\mathscr{I}}$ called the domain containing all the possible resources and resource configurations in our domain. We propose that a design model depicting a number of resource and behavioral diagrams is interpreted as a number of subsets of $\Delta^{\mathscr{I}}$ representing each *resource definition* and each state in the model and as a number of conditions that need to be satisfied by these sets.

A *resource definition* is represented by a set R, such that $R \subseteq \Delta^{\mathscr{I}}$. A resource r belongs to a *resource definition* R iff $r \in R$. We also represent each state S in a statechart as a subset of our domain $S \subseteq \Delta^{\mathscr{I}}$. In this interpretation, the state set S represents all the resources in the domain that have such state active, that is, resource r is in state S iff $r \in S$.

Since resource and behavioral models are represented with class and state machine diagrams respectively, other elements that can appear in a UML model such as generalization of classes, association of classes, state hierarchy and state invariants are interpreted as additional conditions over the sets representing resources and states. For example specialization is interpreted as a condition stating that the set representing a subresource is a subset of the set representing its superresource. These conditions are described in detail in the next section.

In this interpretation, the problem of design model consistency is then reduced to the problem of satisfiability of the conjunction of all the conditions derived from the model. If such conditions cannot be satisfied, then a design model will describe one or more *resource definitions* that cannot be instantiated into resources or resources that cannot ever enter a state in the behavioral model. This can be considered a design error, except in the rare occasion that a designer is purposely describing a system that cannot be realized.

6.2.1 Reasoning Tool Chain

In order to determine the satisfiability of the concepts represented in our design model, we propose to represent the resource and behavioral models using a Description Logic, and analyze the satisfiability of the concepts using an automated reasoning tool. We have chosen OWL 2 DL to represent our UML models since we consider it is well supported and adopted, and there exist several OWL 2 reasoners for checking concept satisfiability.

A number of resource models, behavioral models and state invariants are taken as an input. All the inputs are translated to the OWL 2 DL, a web ontology language [93]. The OWL 2 translation of design models are passed to a reasoner. The reasoner provides report of unsatisfiable and satisfiable concepts. Unsatisfiable concepts will reveal *resource definitions* that cannot be instantiated or behavioral states that cannot be entered.

We have also implemented tool that generates a) skeleton of REST web services from design models, b) OWL 2 DL from design models. The generation of REST web service skeleton is implemented using python in Django web framework [66] which is explained in Chapter 9. The tool that generates OWL 2 DL from design models is discussed later in Section 6.5.

6.3 Description Logic and OWL 2

The Description Logic used in our approach is classified as *SROIQ* [69]. Description Logic is made up of concepts, denoted here by C, D, and roles, denoted here by R, Q. A concept or role can be named, also called atomic, or it can be composed from other concepts and roles.

An interpretation $\mathscr{I}$ consists of a non-empty set $\Delta^{\mathscr{I}}$ and an interpretation function which assigns a set $C^{\mathscr{I}} \subseteq \Delta^{\mathscr{I}}$ to every named concept C and a binary relation $R^{\mathscr{I}} \subseteq \Delta^{\mathscr{I}} \times \Delta^{\mathscr{I}}$ to every named role R.

The constructors of Description Logic are as follows:

Everything		$\top^{\mathscr{I}} = \Delta^{\mathscr{I}}$
Nothing		$\bot^{\mathscr{I}} = \emptyset$
Complement	$(\neg C)^{\mathscr{I}}$	$= \Delta^{\mathscr{I}} \backslash C^{\mathscr{I}}$
Inverse	$(R^{-})^{\mathscr{I}}$	$= \{(y,x) \mid (x,y) \in R^{\mathscr{I}}\}$
Intersection	$(C \sqcap D)^{\mathscr{I}}$	$= C^{\mathscr{I}} \cap D^{\mathscr{I}}$
Union	$(C \sqcup D)^{\mathscr{I}}$	$= C^{\mathscr{I}} \cup D^{\mathscr{I}}$
Restriction		
Universal	$(\forall R.C)^{\mathscr{I}}$	$= \{x \mid \forall y.(x,y) \in R^{\mathscr{I}} \rightarrow y \in C^{\mathscr{I}}\}$
Existential	$(\exists R.C)^{\mathscr{I}}$	$= \{x \mid \exists y.(x,y) \in R^{\mathscr{I}} \wedge y \in C^{\mathscr{I}}\}$
Cardinality	$(\geq nR)^{\mathscr{I}}$	$= \{x \mid \#\{y \mid (x,y) \in R^{\mathscr{I}}\} \geq n\}$
	$(\leq nR)^{\mathscr{I}}$	$= \{x \mid \#\{y \mid (x,y) \in R^{\mathscr{I}}\} \leq n\}$

where $\#X$ is the cardinality of X. Axioms in DL can be either inclusions $C \sqsubseteq D$,$R \sqsubseteq Q$ or equalities $C \equiv D$, $R \equiv Q$.

An interpretation satisfies an inclusion $C \sqsubseteq D$ if $C^{\mathscr{I}} \subseteq D^{\mathscr{I}}$ and an inclusion $R \sqsubseteq Q$ if $R^{\mathscr{I}} \subseteq Q^{\mathscr{I}}$. An interpretation satisfies an equality $C \equiv D$ if $C^{\mathscr{I}} = D^{\mathscr{I}}$ and an equality $R \equiv Q$ if $R^{\mathscr{I}} = Q^{\mathscr{I}}$. $\mathscr{I}$ satisfies a set of axioms if it satisfies each axiom individually – $\mathscr{I}$ is then said to be a model of the set of axioms. Given a set of axioms $\mathscr{K}$, a named concept C is said to be satisfiable if there exists at least one model $\mathscr{I}$ of $\mathscr{K}$ in which $C^{\mathscr{I}} \neq \emptyset$. A set of axioms is said to be satisfiable if all of the named concepts that appear in the set are satisfiable. If a set of axioms $\mathscr{K}$ is satisfiable, we say that an axiom ϕ is satisfiable (with respect to $\mathscr{K}$) if $\mathscr{K} \cup \{\phi\}$ is satisfiable. Similarly, we say that ϕ is unsatisfiable (w.r.t. $\mathscr{K}$) if $\mathscr{K} \cup \{\phi\}$ is unsatisfiable.

The decidability of *SROIQ* is demonstrated by Horrocks et al. [69] and there exist several reasoners that can process answer satisfiability problems automatically [118, 108, 124].

6.3.1 OWL 2 Functional Syntax

For practical reasons, we use the OWL 2 functional syntax (OWL2fs) [93] as the language used as an input for the reasoners and in the text of this article. The interpretation of the main OWL 2 expressions used in this article is presented in the following table. A complete description of the semantics OWL 2, including support for data types can be found in [29].

`SubClassOf(C1 C2)`	$C_1 \sqsubseteq C_2$
`EquivalentClasses(C1 C2)`	$C_1 \equiv C_2$
`DisjointClasses(C1 C2)`	$C_1 \sqcap C_2 = \emptyset$
`ObjectPropertyDomain(P C)`	$\forall R^{-1}.C$
`ObjectPropertyRange(P C)`	$\forall R.C$
`ObjectMinCardinality( n P)`	$\geq nR$
`ObjectMaxCardinality( n P)`	$\leq nR$
`ObjectExactCardinality(n P)`	$(\geq nR) \sqcap (\leq nR)$

In the next section, we discuss and translate the structure of a resource and behavioral model with state invariants over the sets representing *resource definitions* and states into OWL 2 DL.

6.4 From Resource and Behavioral Diagrams to OWL 2 DL

In order to check the satisfiability of *resource definitions* in a resource model and state invariants in behavioral model, we need to first translate all *resource definitions* and their associations into OWL 2 ontology, and then validate the OWL 2 ontology using an OWL 2 reasoner. In this section we present the translation of concepts of resource model and behavioral model to OWL 2.

6.4.1 Resource Model in OWL 2

Each resource in a resource model is shown as a class in an ontology and an association as an object property. A class in OWL 2 is a set of individuals and *ObjectProperty* connect pair of individuals[93]. According to the definition of a resource model given in Definition 1, we need to map these concepts in OWL 2 DL: *resource definitions* and their specializations, attributes, associations and association multiplicities.

Resource Specification and Hierarchy

A *resource definition* in a resource model represents a collection of resources which share same features, constraints and definition. For each *resource definition* in resource model, we define an OWL 2 axiom: `Declaration(Class(R_def))`

Resource model can have resource hierarchy in which subresources of a resource inherit the properties and attributes of its parent resource. We explicitly define the hierarchy of resources in OWL 2 between resources. The specialization of resources represented as classes is reduced to the set inclusion. We represent the fact that a *resource definition* R1 is a specialization of *resource definition* R2 with the condition $R1 \subseteq R2$. In this case we say that R2 is a super resource of R1, analogous to superclass in UML class diagram. If two *resource definitions*

R1 and R2 have a common super resource, or R2 is the super resource of R1 we say that they are in a specialization relation. The specialization relation $R_1 \subseteq R_2$ is translated in OWL 2 as:

```
SubClassOf( R1 R2 )
```

Each *resource definition* at the same hierarchical level in resource model represents a different piece of information. We assume that a resource cannot belong to two *resource definitions*, except when these two *resource definitions* are in a specialization relation. In our semantic interpretation of a resource diagram, it is equally important to denote the facts that two *resource definitions* are not in a specialization relation. We represent the fact that two resources R1 and R2 are not in a specialization relation with the condition $R1 \cap R2 = \emptyset$. With this condition, a resource cannot belong to these two *resource definitions* simultaneously. Due to the open-world assumption used in Description Logic, we need to explicitly state this fact in OWL 2, i.e., for *resource definitions* R1...Rn at same hierarchical level we define disjointness in OWL 2 as:

```
DisjointClasses( R1..Rn )
```

Attributes

In our resource model, a collection resource does not have any attribute and a normal resource should have at least one attribute. So we define attribute *att* of a normal resource *r* of type *D* as $DataProperty(att)$ with domain as *r* and range as *D*. We do not need to give any attribute definition for collection resources because OWL 2 has open world assumption and that which is not mentioned is not considered. So by simply not mentioning collection resources with any attributes is sufficient. However, for every normal resource, each of its attributes is defined in OWL 2. Attributes usually have a multiplicity restriction to one value. Hence, the attribute definition *att* in OWL 2 is given as:

```
Declaration(DataProperty( att ))
SubClassOf(C DataExactCardinality(1 att ))
DataPropertyDomain( att r )
DataPropertyRange( att D )
```

Association

An association *a* is a relation between two *resource definitions*, r_1 and r_2 and name of association is its label *l*, i.e., *l(a)*. For each association *a* in the resource model, we define `ObjectProperty` axiom with label *l* and r_1 as its domain and r_2 as its range,i.e., for $a(r_1, r_2)$ with $l(a)$, we give OWL 2 definition as:

- *l* maps to OWL 2 axiom `Declaration(ObjectProperty(l))`
- r_1 maps to OWL 2 axiom `ObjectPropertyDomain(l` r_1`)`

- r_2 maps to OWL 2 axiom `ObjectPropertyRange(l` r_2`)`

We define $min(a)$ and $max(a)$ as minimum and maximum cardinality of association a. It defines the number of allowed resources that can be part of the association. We represent a directed binary association A from *resource definition* R1 to R2 as a relation $A : R_1 x R_2$. The multiplicity of the association defines additional conditions over this relation $\#\{y|(x,y) \in A\} \geq min$, $\#\{y|(x,y) \in A\} \leq max$. If an association a has minimum cardinality *min* and maximum cardinality *max* for resource r, we define it in OWL 2 as:

```
SubClassOf( r ObjectMinCardinality( min a ) )
SubClassOf( r ObjectMaxCardinality( max a ) )
```

6.4.2 Behavioral Model in OWL 2

A behavioral model provides the behavioral interface of a web service and defines the sequence of method invocations, the conditions under which they can be invoked and their expected results. To check the satisfiability of state invariants in a behavioral model, we need to translate the states and their invariants into OWL 2. The translation of the state and the state invariant includes the reference of resources and their attributes so we translate a behavioral model in the same ontology that contains the OWL 2 translation of a resource model.

We need to cover following concepts of our behavioral model in OWL 2: state, state hierarchy, state disjointness and state invariant.

State and State Hierarchy

A state in the behavioral model is a concept that defines the state of the service. The behavioral model results in emergence of many new concepts in our ontology. Each state represents a piece of information and can be exposed as an ontology class.

```
Declaration(Class(S))
```

State hierarchy is also represented using set inclusion. Whenever a substate is active, its containing state is also active. This implies that if a resource belongs to a set representing the substate, it will also belong to the set representing the super state, $sub \subseteq S$. We define each substate relationship explicitly in OWL 2. For each state *sub* that is a substate of composite state *s*, we define OWL 2 axiom as:

```
SubClassOf( sub s )
```

The states at same hierarchical level represent different resource configurations such that only one state can be active at same time. The composite states with non-orthogonal regions also follow the same principle, i.e., only one state can be active at same hierarchical level. This means that a resource cannot be at the same

time in the two sets representing two exclusive states, i.e., if S_1 and S_2 represent substates of an active and not orthogonal composite state then $S_1 \cap S_2 = \emptyset$. When representing a state machine in OWL 2, the non-orthogonal exclusive states are declared as disjoint, so that they may not able to share any resource.

```
DisjointClasses( S1..Sn )
```

In a composite state with orthogonal regions, two or more states can be active at the same time if they belong to two or more different regions of composite state, i.e, if R_1 and R_2 are the regions of an active and orthogonal composite state S then $R_1 \cup R_2 = S$. We should note that if $region(s_1) \neq region(s_2)$ then they are not exclusive and $S_1 \cap S_2 \neq \emptyset$. Due to the open-word assumption of DL we do not need to define this non-exclusiveness since classes may represent same set of instances unless they are explicitly declared as disjoint.

6.4.3 State invariant into OWL 2 DL

The invariant condition characterizes the state, i.e., if the invariant condition holds, then the state is active and if the invariant condition does not hold, then the state is not active.

In our approach we represent an invariant as a set of resources that makes that invariant evaluate to true. Since the invariant holds iff the associated state is active, the set representing a state will be the same as the set representing an invariant. This is represented in OWL 2 as an equivalent class relation between the state and its invariant:

```
EquivalentClasses(S Invariant)
```

Due to the equivalent relationship between state and its invariant, all resources that fulfill the condition of its state invariant will also be in that specific state.

State Constraints

Our behavioral model is represented by a UML protocol state machine with additional constraints. The UML allows us to define additional constraints to a state, and names these constraints as state invariants. However, the semantics of a state constraint is more relaxed since it "specifies conditions that are always true when this state is the current state" ([125], p.562). In this sense, the state constraints define necessary conditions for a state to be active, but not sufficient. This means that, the actual state invariant may remain implicit. However, we consider a state invariant as a predicate characterizing a state. That is, a state will be active if and only if its state invariant holds.

The UML Superstructure specification requires that whenever a state is active its state invariant evaluates to true ([125],p.562). A consequence of this is that state invariants should be satisfiable. That is, every state invariant in a state machine must

⟨OCL-expression⟩	::=	⟨cond-expr⟩ (⟨logic-op⟩⟨cond-expr⟩)*
⟨logic-op⟩	::=	*and* | *or*
⟨cond-expr⟩	::=	⟨ref⟩ →size()⟨relational-operator⟩⟨integer-literal⟩
		| ⟨ref⟩ →isEmpty() | ⟨ref⟩ → notEmpty()
		| ⟨ref⟩⟨relational-operator⟩⟨primitive-literal⟩
⟨ref⟩	::=	self.⟨identifier⟩
⟨identifier⟩	::=	′{⟨characters⟩} | 0..9 {0..9}′
⟨relational-operator⟩	::=	< | <= | > | >= | <> | =
⟨primitive-literal⟩	::=	⟨boolean-literal⟩ | ⟨integer-literal⟩
		| ⟨string-literal⟩ | null
⟨boolean-literal⟩	::=	true | false
⟨integer-literal⟩	::=	0..9 {0..9}
⟨string-literal⟩	::=	′{⟨characters⟩}′

Figure 6.3: The grammar of the supported OCL fragment.

hold in at least one resource configuration. Otherwise there cannot be resources that have such state active. Since invariants should be satisfiable, the set of resources S representing a state should not be empty $S \neq \emptyset$.

6.4.4 State Constraints in μ OCL

A state invariant is a runtime constraint on the state ([125],p.514). It is used to express a number of constraints, such as the restriction on the values of resource attributes or the restriction on the existence of resources by using the multiplicity constraint of the associations. These constraints are combined by using boolean operators.

We have used a subset of OCL to define state invariants in behavioral models of REST web services that are represented by UML state machine diagrams. Unfortunately, in general OCL is not decidable. However, we can avoid undecidability by restricting our approach to a reduced fragment of the full OCL [106]. The use of a limited fragment of OCL to avoid undecidability has been proposed in the past also by other authors [106, 107].

In this work, we consider OCL constructs using mainly multiplicity, attributes value and boolean operators. The grammar of OCL, supported in our approach is shown in Figure 6.3.

Attribute Constraints

The OCL attribute value constraint $self.Att = Value$ is mapped to OWL 2 axiom *DataHasValue* as follows:

```
DataHasValue(Att "Value"^^datatype )
```

where *Att* is the name of the attribute, *Value* is the value of the attribute, and *datatype* is the datatype of the attribute *Value*.

Multiplicity Constraints

The translation of *size*() operation in OWL 2 is based on the infix operator used with the *size*() operation, such as:

- "*size*() >=" or "*size*() >" is mapped to OWL 2 axiom: *ObjectMinCardinality*
- "*size*() <=" or "*size*() <" is mapped to OWL 2 axiom: *ObjectMaxCardinality*
- "*size*() =" is mapped to OWL 2 axiom: *ObjectExactCardinality*

For example, the OCL constraint $self.A->size() = Value$, in which *A* is the name of an association and *Value* is a positive integer, is written in OWL 2 as:

```
ObjectExactCardinality(Value A)
```

Boolean Operators

The constraints in a state invariant are written in form of a boolean expression, and joined by using the boolean operators, such as "*and*" and "*or*" ([96],p.144).

- The binary "*and*" operator evaluates to true when both boolean expressions Ex_1 *and* Ex_2 are true. In our translation this is represented by the intersection of the sets that represent both expressions $Ex_1 \cap Ex_2$. This is represented OWL 2 as `ObjectIntersectionOf(Ex1 Ex2).`
- The binary "*or*" operator evaluates to true when at least one of the boolean expression Ex_1 *or* Ex_2 is true. In our translation this is represented by the union of the sets that represent both expressions $Ex_1 \cup Ex_2$. This is represented OWL 2 as `ObjectUniounOf(Ex1 Ex2)`

6.5 Consistency Analysis using an OWL 2 Reasoning Tool

We have defined earlier the satisfiability of our design models in Sect. 6.2. The consistency analysis of resource and behavioral models is reduced to the satisfiability of the conjunction of all the conditions derived from the model. In order to determine the satisfiability of the conditions represented in design models, we first translate the resource and behavioral models into an OWL 2 ontology, then use an OWL 2 reasoner to analyze the satisfiability of translated concepts.

To translate the resource and behavioral models into OWL 2 ontology, we have implemented the translations of resource and behavioral diagrams in OWL 2, discussed in Sect. 6.4, in form of a translation tool. We have used Python programming language for the implementation of the prototype of the translation tool. The implemented translation tool allows us to automatically transform a resource and behavioral model into OWL 2 DL. The translator takes these models and μOCL state invariant as an input in the form of XMI. The XMI is generated by using

```
// Resource Model into OWL 2 DL
Declaration(Class(collectionbookings)
Declaration(Class(Booking))
Declaration(Class(Room))
...
DisjointClasses( collectionbookings Room Cancel ...)
Declaration(ObjectProperty(cancel))
ObjectPropertyDomain( cancel Booking)
ObjectPropertyRange( cancel Cancel )
...
SubClassOf( Booking
ObjectMaxCardinality( 1 cancel ))
....
Declaration(DataProperty( guestName))
SubClassOf(Booking
DataExactCardinality(1 guestName))
...
DataPropertyDomain(guestName Booking )
..
DataPropertyRange(guestName xsd:string )
..

//Behavioral Model into OWL 2 DL
Declaration(Class(activeBooking))
Declaration(Class(canceled))
SubClassOf( activeBooking HotelRoomBooking )
SubClassOf( canceled HotelRoomBooking)
...
DisjointClasses( activeBooking canceled)
Declaration(Class(notConfirmed))
SubClassOf(notConfirmed activeBooking)
Declaration(Class(notpaid))
Declaration(Class(processingpayment))
SubClassOf( notpaid notConfirmed)
SubClassOf( processingpayment notConfirmed )
DisjointClasses(notpaid processingpayment)
...
//Invariant of state confirmed Start
EquivalentClasses (confirmed
ObjectIntersectionOf(
ObjectExactCardinality(1 payment)
ObjectExactCardinality(1 confirmation)
ObjectExactCardinality(0 processing))
//Invariant of state confirm End
```

Figure 6.4: The excerpt of the output ontology generated by the translation tool.

a modeling tool, Magicdraw. The XMI generated by the modeling tool contains the source code of both resource and behavioral model in form of XML. While modeling in a modeling tool we have used μOCL to express the state invariants in a behavioral model. The state invariant written in μOCL is also part of a XMI generated by the modeling tool. Moreover, the output of the implemented translation tool is an ontology file, which contains the transformed resource model, behavioral model and state invariants in form of OWL 2 functional syntax. This output file is ready to be processed by an OWL 2 reasoner.

As an example, we have translated the resource model, behavioral model and OCL state invariants shown in Figure 6.1 and Figure 6.2, into OWL 2 DL ontology using the implemented translation tool. An excerpt of the output ontology generated by the translation tool is shown in Figure 6.4.

6.5.1 Reasoning

After translating the resource model, behavioral model and state invariants into OWL 2 ontology by using the implemented translation tool, we validate the output ontology by using an OWL 2 reasoner. The OWL 2 reasoner analyzes different facts presented as axioms in the ontology and infers logical consequences from them. When we give the generated ontology to the reasoner, it generates satisfiability report indicating which concepts are satisfiable and which not. If the ontology has one or more unsatisfiable concepts, this means that the instance of any unsatisfiable concept will make the whole ontology inconsistent, consequently, an instance of the resource describing an unsatisfiable concept in a resource model will not exist, or resources will not enter in a state describing an unsatisfiable condition.

In order to analyze the satisfiability of the invalid invariants, the ontology of an example model with invalid invariants, listed in Section 6.1.1, is validated by using an OWL 2 reasoner name Pellet [118]. The satisfiability report of the ontology of service design models with invalid state invariants is shown in Figure 6.5. As explained in Section 6.1.1, the introduced errors in the state invariants of state *processingpayment* resulted in 4 unsatisfiable concepts. This is because invariants of non-orthogonal states should be mutually exclusive. By changing the invariant clause $payment.processing->size() = 1$ of *processingpayment* to $payment.processing->size() = 0$, the non-orthogonal states *processingpayment* and *notpaid* became inconsistent, making the super state *activeBooking* also inconsistent. Similarly, whenever a state is active, all its superstates should be active. When invariant clause `self.cancel->size()=1` is introduced in the invariant of the state *processingpayment*, it contradicted with invariant of its superstate *activeBooking* resulting in conflict with the invariant of state *canceled*. Thus, making state *canceled* inconsistent as well.

As explained in Section 6.4.3, a state invariant characterizes the state ([125], p.559-560). Therefore, the presence of unsatisfiable states in the satisfiability report indicates the existence of invalid state invariants in identified states.

```
Found 4 unsatisfiable concept(s):
a:processingpayment
a:notpaid
a:activeBooking
a:canceled
```

Figure 6.5: The satisfiability report of the ontology shown in Figure 6.4 generated by the OWL 2 reasoner Pellet.

6.5.2 Performance Test

In order to determine the performance of the translation tool and reasoning engines, we conduct a number of tests by using UML class and statechart diagrams consisting of 10 to 2000 model elements. The performance tests are conducted for both valid and mutated models. These tests are evaluated on the bases of UML to OWL 2 translation time, and the reasoning time taken by OWL 2 reasoners Pellet [118] and HermiT [108]. The performance test results are shown in Table 6.1, and the total time (Translation time + Reasoning time) to process UML models is shown in Figure 6.6.

Table 6.1: Time taken by the translation tool and reasoning engines to process UML models.

Model Elements	10	100	500	1000	1500	2000
Translation Time	0.08s	0.11s	0.19s	0.30s	0.44s	0.53s
Pellet						
Valid	2.2s	2.3s	2.6s	3.2s	3.6s	3.8s
Mutated	2.2s	2.4s	2.7s	3.2s	3.6s	3.9s
HermiT						
Valid	0.6s	0.7s	1.2s	1.7s	2.2s	2.6s
Mutated	0.7s	0.7s	1.3s	1.8s	2.3s	2.6s

The complexity of OWL 2 DL with respect to the reasoning problems of ontology consistency and instance analyzing is NEXPTIME complete [70]. However, the graph (Figure 6.6) of the performance test shows the linear curve, because in our approach we analyze the consistency of class and statechart diagrams without individuals.

6.6 Related Work

Consistency analysis and checking of design models has been studied by number of researchers in the past but in the area of web services it has not been researched very extensively, especially in the area of REST web services, we were unable to

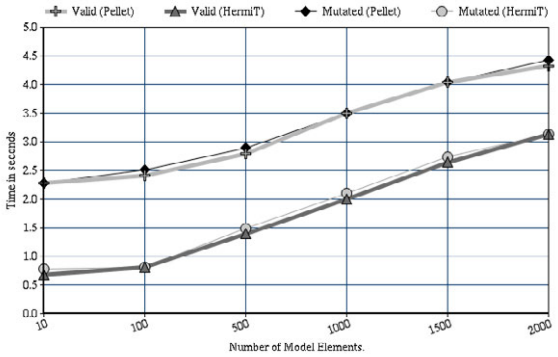


Figure 6.6: The graph of the total time (Translation time + Reasoning time) to process valid and mutated models.

find any consistency checking approaches. However, in the area of consistency checking for web services following works are noteworthy.

Yin et al. [129] use type theory to verify consistency of web services behavior. The paper addresses web services choreography. It analyzes structure of service behavior and uses extended MTT, which is a constructive type theory, to formally describe service behavior. The procedures of deductions are then given that verify the suitability between services along with discussion on type rules for subtype, duality and consistency of web services behavior.

Tsai et al. [123] [122] present a specification based robust testing framework for web services. The approach first uses an event driven modeling and specification language to specify web services and then uses a completeness and consistency (C & C) approach to analyze them. Based on these positive and negative test cases are generated for robustness testing. The approach assumes that web services are specified in OWL-S. The approach aims towards testing of web services but C&C analysis is performed on OWL-S specification that was point of interest for us. The approach identifies missing conditions and events, whereas, our approach checks the structure of web services and validates implementation of service requests. Xiaoxia [40] verify the service oriented requirements using model checking. The service-oriented computer independent model is used to structure the requirements and then automated model checking is done to do completeness and consistency checking of requirements. It provides formal definition of completeness checking as a check that all the required service are included in model and does not give any specific consistency checking constraint except the requirement relation applied on an example.

Nentwich et al. [94] present a static consistency checking approach for distributed specifications by describing xlinkit framework. This provides a distribution-transparent language for expressing constraints between web service specifications. The implementation of xlinkit is done on light-weight web service using XML.

Heckel et al. [64] present a model-based consistency management approach

for web service architectures. They advocate use of UML class and activity diagrams for modeling web services. The consistency problems are then identified in UML based development process. For each consistency problem a partial formalization into a suitable semantic domain is done and a consistency check is defined. The consistency problems identified include syntactic correctness and deadlock freedom. Based on these, those activity diagrams are identified that are relevant to consistency checks. These are partially translated to CSP which are then assembled in a single file and handed over directly to model checker. The paper outlines a good consistency management approach for web services architecture that needs concrete development of different steps defined in the paper.

6.7 Conclusion

A REST interface can do more than simply creating, retrieving, updating and deleting data from a database. Designing behavioral interface for such web services that provide different states of the service and offer REST interface features is an interesting design challenge since it can involve many resources and resource configurations that define different states of the service. In this chapter, we address how to analyze the consistency of design models that create behavioral REST interfaces. We check the consistency of resource and behavioral diagrams with state invariants using OWL 2 reasoners. The structure of both the diagrams are translated to OWL 2 ontology and ontology reasoners are used to check any unsatisfiable concepts in the ontologies. The unsatisfiable concepts indicate the design errors that can cause undesirable behavior in the implementation of the service. The approach is automated as we provide prototype tools that generate web service skeletons and OWL 2 ontology from the design models. Also, thanks to the existing OWL 2 reasoners the generation of satisfiability report for our OWL 2 ontology is also automated that is analyzed to check the consistency of design models.

Chapter 7

Web Service Composition

A web service composition uses existing web services to provide new functionality built on top of them. It facilitates reusability of already published web services and combines them into a whole to serve different purposes. These different services may be developed by different vendors in different locations. Service orchestration is a common approach for developing a composite web service. In services orchestration, the involved services are unaware of their participation in the composition process and a central process controls and coordinates the execution of these services.

For big web services, different flow composition languages exist like BPEL4WS [19] and WSCL [22]. These languages define the control and data flow which determines when a certain operation should execute. Business Process Execution Language for Web Services (BPEL4WS) [19] is one of the composition specification languages that is widely adopted to implement a web service composition. In addition, many modeling approaches have also been proposed for composition of SOAP based web services [116] [109].

However, REST web services follow a different architectural style and thus require different design philosophy and techniques. A web service composition in RESTful style differs from traditional web service composition techniques since instead of composing web services from the perspective of operations, RESTful composition focuses on resources [132]. A RESTful web service takes resource as a building block and in a typical REST development environment, new resources are exposed to keep the design simple and to allow maximum decoupling. In addition to offering REST interface features, i.e., *addressability*, *connectivity, statelessness* and *uniform interface*, designing a REST composite web service must take the method invocations on partner services in account and vice versa. Web service compositions may also offer time critical behavior that should be taken into account.

In this chapter, we show how the composition of web services is done in RESTful manner. The background of composition technologies is given in section

7.2. An overview of our composition approach is given in section 7.3. The static structure of composite REST web service is presented as resource model in section 7.4. Section 7.5 details the construction of process modeling for REST composite service and section 7.6 shows how its REST interface is constructed. The related work and conclusion are given in sections 7.7 and section 7.8.

7.1 Background

Web service compositions are often assumed to be driven by a business goal [119] and are described separately in a flow specification language. These composite web services are like a black box to the end user and only advertise the operations that can be invoked on them through the interface. The specification of an interface only contain the syntactic information about the operations that can be invoked on them and the flow in which messages can be exchanged between the services is described separately in a flow specification language [119].

Web Service Description Language (WSDL) [44] is often used for the interface specifications of big web services. WSDL provides information on the operations that a service allows and defines the data that is transmitted as messages between service operations. The information on the order of method invocation is defined in a flow specification language like BPEL4WS[19]. However, we cannot use the existing tools and techniques to compose REST web services since REST web service APIs do not use standard WSDL [102]. Also, the uniform interface principle of REST does not fit well with the message oriented constructs of BPEL4WS.

The following listing presents three possible mechanisms for defining RESTful BPEL4WS processes:

1. **WSDL 2.0 for HTTP binding**: Using normal BPEL4WS [19] descriptions (e.g. invoke operation1) and funneling invocation through a synthetic WSDL 2.0 description. In WSDL binding, operations are mapped to URIs and HTTP methods (e.g operation1 maps to resource1 and PUT method).
2. **REST through adapter**: Using RESTful services from a normal BPEL4WS process by generating an artificial WSDL which describes the REST interface and then creating an adapter for communicating with the RESTful service. Some BPEL4WS engines also support interaction through WADL description.
3. **BPEL extension for REST**: Extending BPEL4WS (and a BPEL engine) for directly supporting RESTful activities (e.g. GET, PUT, POST, and DELETE activities in a scope of a certain resource).

Approaches 1 and 2 have advantage of not requiring any modification to BPEL4WS language and current implementations of BPEL4WS engines. On the other hand, WSDL is a description language for operation-oriented web services and it does not fully comply with the semantics of resource-oriented RESTful services. In [102, 100], Pautasso uses the third approach and proposes extensions

to BPEL for REST. This approach has several advantages, such as, supporting a late binding and dynamic resources. In traditional BPEL4WS [19] processes, dynamic binding of services is not supported, as service endpoints are predetermined and any changes requires clients to change or update their WSDLs. Thus, this is a significant advantage of the third approach. Here we summarize the main differences in BPEL4WS and BPEL-REST, as proposed by C. Pautasso in [102, 100].

- In BPEL4WS only one invoke type of activity is declared, where as in BPEL-REST for each HTTP operation a distinct invoke activity is defined.
- BPEL-REST introduces a resource concept, which allows defining and publishing resources. It defines allowed message handlers, which are available in the scope of the resource.
- The internal behavior of a request handler can be specified using the normal BPEL4WS constructs (i.e.,
  $< sequence >$, $< if >$, $< flow >$, $< while >$, etc.).
  However, control-flow links across activities that belong to different handlers are not allowed.

We use the information presented in this section about RESTful web service composition to model the composition of REST web services. In the next section, we provide an overview of our modeling approach using UML for designing composite RESTful web services.

7.2 Overview

Our approach takes as input the behavioral interface specifications of the partner REST web services and the business requirements. The composite REST web service interface is modeled with a resource model, behavioral model and a class diagram representing the domain model. The process of composition is elaborated with the sequence and activity diagrams. The sequence diagrams can be created from the specification document or alternatively from the service requirements. The service requirements are derived from the specification document and can also be labeled as comments on the behavioral model as shown in Chapter 4 or they can be modeled as different scenarios using sequence diagrams. We have used scenario modeling to model different requirements in this chapter to provide a step-wise approach to construct process model as proposed in [113]. On the other hand, labeling of requirements on the behavioral model can facilitate requirement coverage during validation phase. Users can use either of these two approaches to capture requirements alternatively or simultaneously, depending on their need.

The resource and behavioral models follow the same design principles detailed in the previous chapters. For composite web service, we extend our behavioral models with effects on transitions for invoking the partner services and also introduce the class diagram as domain model to show the provided and required interfaces between the composite service and its partner services.

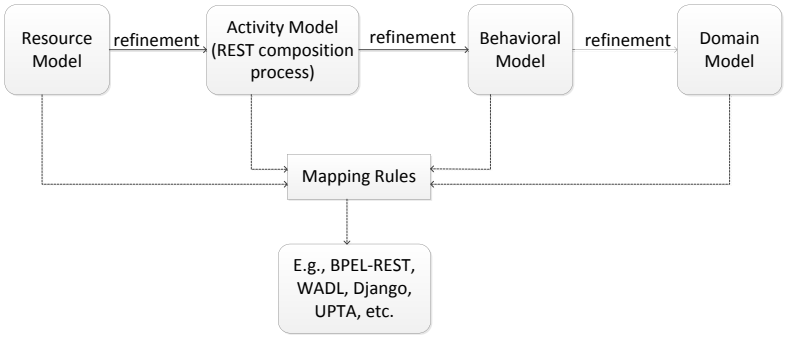


Figure 7.1: Approach for Model-based RESTful composition

We start with the resource model of a composite RESTful web service. The composite behavior of the service is then sketched as a sequence diagram and the process flow is modeled as an activity diagram. The process information is refined to behavioral models that give information on how the REST composite web service interface is defined. This constraints the implementation and usage of composition to provide RESTful behavior. Our modeling approach is shown in Figure 7.1 and provides mapping rules for translation to different implementation and specification languages.

We assume that the partner services are RESTful and their design models are available or they can be constructed before using them in the composition process. This assumption helps us in providing RESTful interface for composite web service. We consider that this is not a strong assumption since partner services need to be understood first in order to know the functionality they offer and make them a part of the composition process. These models are considered part of the specification document of web services and do not require any extra effort from the Composite RESTful Web Service (CRWS) developer. Alternatively, the developer can also design the REST service composition models based on the specification document even if the partner web services are not specified with our approach.

In order to exemplify our approach, we take a Holiday Booking (HB) REST composite web service (CWS) that is built on inspiration from the *housetrip.com* service. It is a holiday rental online booking site, where you can search and book an apartment in the country of your destination. We have built it as a REST composite web service.

The user of the service searches for a room in a hotel from the list of available hotels at holiday booking service before travel. He books the room (if it is available) and that booking is reserved by holiday booking service with the hotel for 24 hours. The user must pay for the booking within 24 hours. If the user does not pay within this time then the booking is canceled. If the user pays, then the holiday booking service invokes a payment service and waits for the payment confirmation. When the payment is confirmed by the payment service, holiday booking service invokes

the hotel service to confirm the booking of the room. If the hotel does not respond within one day or it does not confirm at all, the booking is canceled and the user is refunded. If the hotel service confirms, then a booking is made with the hotel. The payment is not released to the hotel until the user checks in. When the user checks in and is satisfied, holiday booking service releases the money to the hotel and booking is marked as paid by the hotel.

7.3 Resource Model

A resource model of a composite RESTful web service follows the design principles highlighted in Chapter 3. To summarize, a resource model is represented by a UML class diagram where each class represents a *resource definition*. The *resource definitions* are instantiated as resources, analogous to the relationship between *class* and its *objects* in object oriented paradigm. The direction of association between *resource definitions* gives the navigability direction between them and the role name gives its URI (addressability). The *collection resource definitions* with no incoming transitions are considered *root resource definitions* for the resource model since every other *resource definition* should be reachable via *root*. The resource model should make a connected graph.

Figure 8.4 shows resource model of our holiday booking composite REST web service. The model has one *collection resource definition*, i.e,. *bookings*, and 13 normal *resource definitions*, i.e., *booking, Pay, Paid, pRelease, ConfirmHPRelease, WaitingPRelease, hotelCheck, hotelConfirm, WaitinCheckIn, CheckInConfirm, cancel, WaitingRefund and Refund*. A GET method can be invoked on each resource. The *root resource definition* in the model is *bookings*. This resource can contain many booking resources. A booking *resource definition* is associated to different *resource definitions* that specify the addressable information (via URI) for that resource. For example, a GET on */bookings/{booking_id}/paid/* returns either a response code of 404 if the booking is not paid or a response code of 200 if the booking is paid along with the resource representation. Table 7.1 gives a clear description of the information represented by each resource definition.

Resource model provides structural layout of composite REST web service. We use this structural model to present our behavioral models in Section 5 and 6. The next section, presents modeling of the RESTful process.

7.4 Modeling a RESTful process

The UML activity diagram and BPMN [128] are the common notations used for BPEL4WS process modeling. A proposal for UML Profile for BPEL4WS [18] uses old BPEL 1.0 specification and UML 1.4, but provides guidelines on how to map BPEL4WS processes to UML activity diagrams. Also, a UML 2.0 profile for BPEL4WS have been proposed [17]. Ruokonen *et al.*, in [113], presents a

Table 7.1: Information represented by resources of Holiday Booking REST CWS

Resource	Information
1- collection_bookings	Gives a list of all the bookings
2- booking	Gives detail of a specific booking
3- Pay	Tells whether the booking payment is being processed by the payment service or not
4- Paid	Tells if a booking is paid or not
5- pRelease	Tells if the payment of the booking is released to the hotel or not
6- WaitingPRelease	Tells whether a payment release is being processed by the payment service or not
7- ConfirmHPRelease	Tells if the payment release is confirmed to the hotel
8- hotelCheck	Tells if the booking is being processed by the hotel for its confirmation
9- hotelConfirm	Tells if the booking is confirmed by the hotel
10- WaitingCheckIn	Tells if the booking is waiting for user check in
11- CheckInConfirm	Tells if the user has checked in or not
12- cancel	Tells if the booking is canceled
13- WaitingRefund	Tells if the booking is being processed by the payment service for refund
14- Refund	Tells if the booking is refunded or not

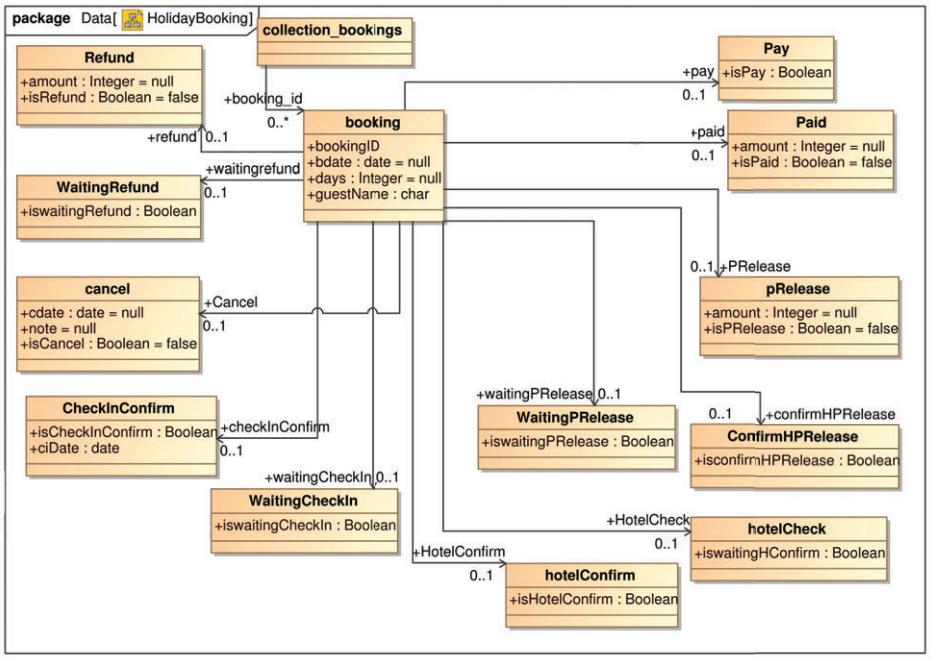


Figure 7.2: Resource Model for HF Composite RESTful Web Service

UML-based approach for creating BPEL4WS flavored activity models, which enable generation of executable BPEL4WS descriptions. By combining existing works, we aim at proposing modeling constructs, given as UML activity diagram, for BPEL-REST processes.

7.4.1 Scenario Models

Our target is to model a RESTful process for Holiday Booking composite REST web service using the static structure of the service defined in the resource model shown in Figure 7.2. We start with a set of sequence diagrams that show a number of interactions with the composite service. A sequence diagram shows the object interactions using time-oriented visualization [103]. We call these scenario models. We require that the modeler sketches simple example scenarios illustrating required behavior of the process. This set of scenario may not (but can) contain all the possible execution scenarios. For the details of this approach presented by Ruokonen et al. readers are referred to [113]. Some exemplary process scenarios for our approach are presented in Figure 7.3. Here, *BookingH* is a process, which is to be implemented as a RESTful composite service. *BookingH* uses two external RESTful services: *PaymentService* and *hotelService*. In Figure 7.3 (top), a customer invokes the composite service *BookingH* and pays for it. When the customer pays, *BookingH* invokes the partner service, *PaymentService*. The service

returns a payment confirmation by invoking the PUT method on *paid* resource of *BookingH*. In the scenario shown in the bottom of Figure 7.3, *BookingH* composite service invokes the partner hotel service, *hotelService*, for booking confirmation. In case, the hotel service confirms the room booking, *BookingH* service waits for user check in. When the user checks in, *BookingH* invokes the payment service to release the payment to the hotel that was withheld. When the payment service confirms the release of the payment, *BookingH* marks that payment for the booking has been released. It, then, also invokes a POST on confirm resource of the hotel service to mark it as paid.

7.4.2 Process Model

From the scenario models, we want to create a process description, which implements these scenarios. We aim at a process model, which is compatible with BPEL-REST process language. For process modeling, we chose UML activity diagram that are used to model the flows in a process [125]. To construct the process model, we model *BookingH* service's view on the conversation. This means messages sent and received by the *BookingH* service. Send message events in a sequence diagram present invocations in an activity diagram and receive message events means that the process receives an operation call.

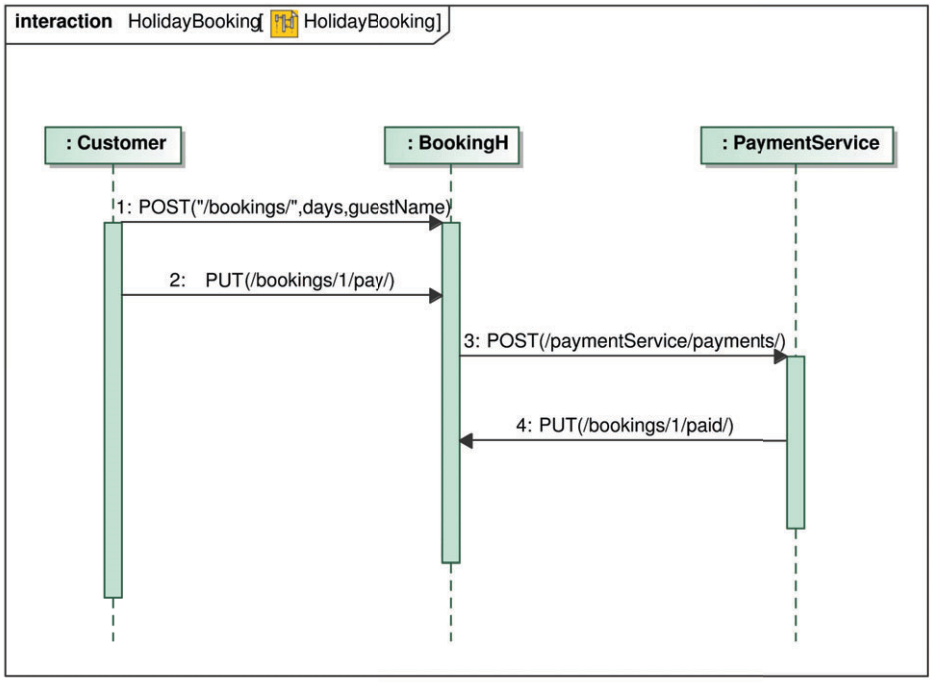
In Figure 7.4, a workflow-oriented model for *Holiday Booking* process is shown as an activity diagram. The following listing presents how UML activity model constructs are used to model BPEL-REST processes. Pautasso [102] proposes four activities to invoke a REST web service from the process, i.e., HTTP invocations *get*, *put*, *post*, and *delete*. Thus,

- Method invocations are modeled as a call behavior action (stereotype $<<$ *get* $>>$, $<<$ *put* $>>$, $<<$ *post* $>>$, $<<$ *delete* $>>$ respectively). A call behavior action in UML activity diagram invokes an activity or a state machine [125].

Request handlers receive the request invoked on the composite service and perform the corresponding tasks. The request handlers in BPEL-REST are defined for each resource and stem from the REST uniform interface principle. Thus, *onPut*,*onGet*,*onPost*, and *onDelete* request handlers are mapped to activities as follows:

- *onPut* is modelled as a call behavior action (stereotype $<<$ *onPut* $>>$)
- *onGet* is modelled as a call behavior action (stereotype $<<$ *onGet* $>>$)
- *onPost* is modelled as a call behavior action (stereotype $<<$ *onPost* $>>$)
- *onDelete* is modelled as a call behavior action (stereotype $<<$ *onDelete* $>>$)
- *respond* is modelled as a call behavior action (stereotype $<<$ *respond* $>>$)
- *sequence* is modelled as a control flow

Unlike *onMessage* activities in BPEL4WS, RESTful message handlers can have internal structure and thus they could be also modeled as structured activities,



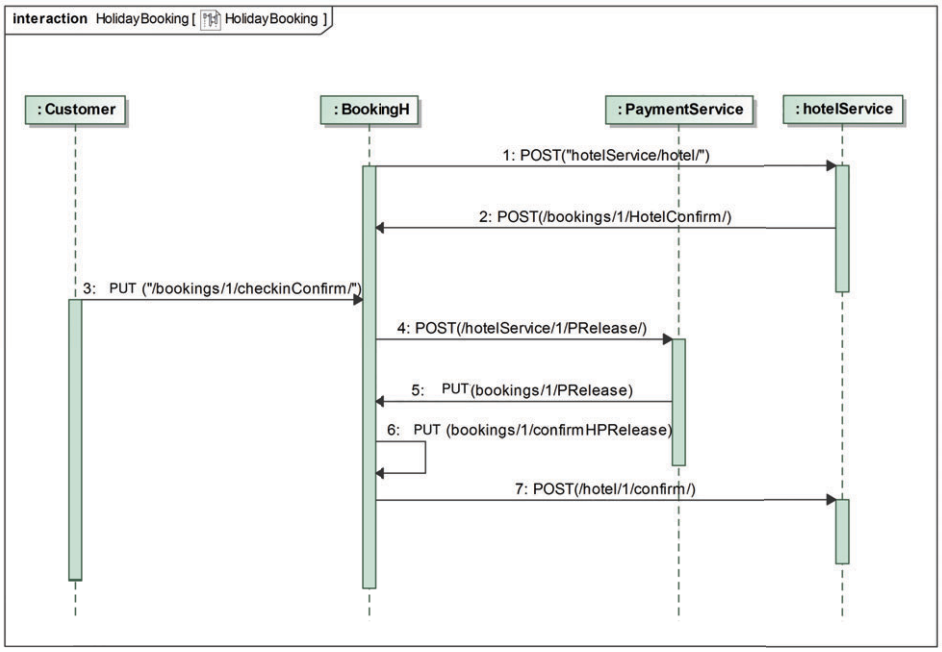


Figure 7.3: Examples of Holiday Booking Scenarios

which contain a *respond* activity and possibly some other if-else structures. On the other hand, *onPut* and *respond* could be modeled as a pair of simple activities, which has a potential sequence of activities between them. The same applies of course to other message handlers. As a modeling restriction, links across activities, which belong to different message handlers are not allowed. A resource in an activity model, would be modeled as a structured activity, which contains all the message handlers it supports. However, our target is to simply model the workflow, which models the desired behavior, in the activity diagram.

The process model defines the workflow of the composition process. In order to look at the web service composition from the viewpoint of how different resource attributes change and the conditions of method invocations, we model the web service composition with UML state machine in the next section.

7.5 Modeling Composite RESTful interface

In this section, we refine the activity diagram that gives flow of activities to a protocol state machine that defines our behavioral model. In the refinement step, a send message (the process makes an invocation) is mapped to an effect of transition in a state machine. A receive message in an activity diagram (the process receives a message) is mapped to a state transition in a state machine.

In behavioral model each state represents a state of the service and trigger methods are restricted to the side-effect methods of HTTP, i.e., PUT, POST and DELETE (uniform interface). Although REST offers a stateless interface but we are able to represent stateful services built in RESTful manner by defining state invariants as predicates over resources using HTTP GET method on resources (statelessness). The construction of behavioral model is detailed in Chapter 4, however, for modeling a service composition, the models are required to represent method invocations on the partner services. These service invocations to partner services are modeled as effects on the transitions.

Figure 7.5 shows the behavioral model for our holiday booking composite REST service and Figure 7.6 shows its domain model. The domain model shows the required and provided interface methods between REST composite web service and its partner services. The behavioral model of partner web services, *PaymentService* and *HotelService*, are shown in Figure 7.7.

The behavioral model of holiday booking service shown in Figure 7.5 is initiated with POST on *bookings* with *days* and *guestname* as parameters. When the user makes a PUT on pay, third party payment service is invoked by the composite service to process users payment. This is shown as an effect on the transition that invokes the *PaymentService* shown in 7.7 (top). While the payment is being processed, the service goes in to a wait state. The payment service responds either by invoking a PUT on paid resource in case of confirmation or by invoking a DELETE method for *pay* resource, in case the payment is not confirmed. An unpaid

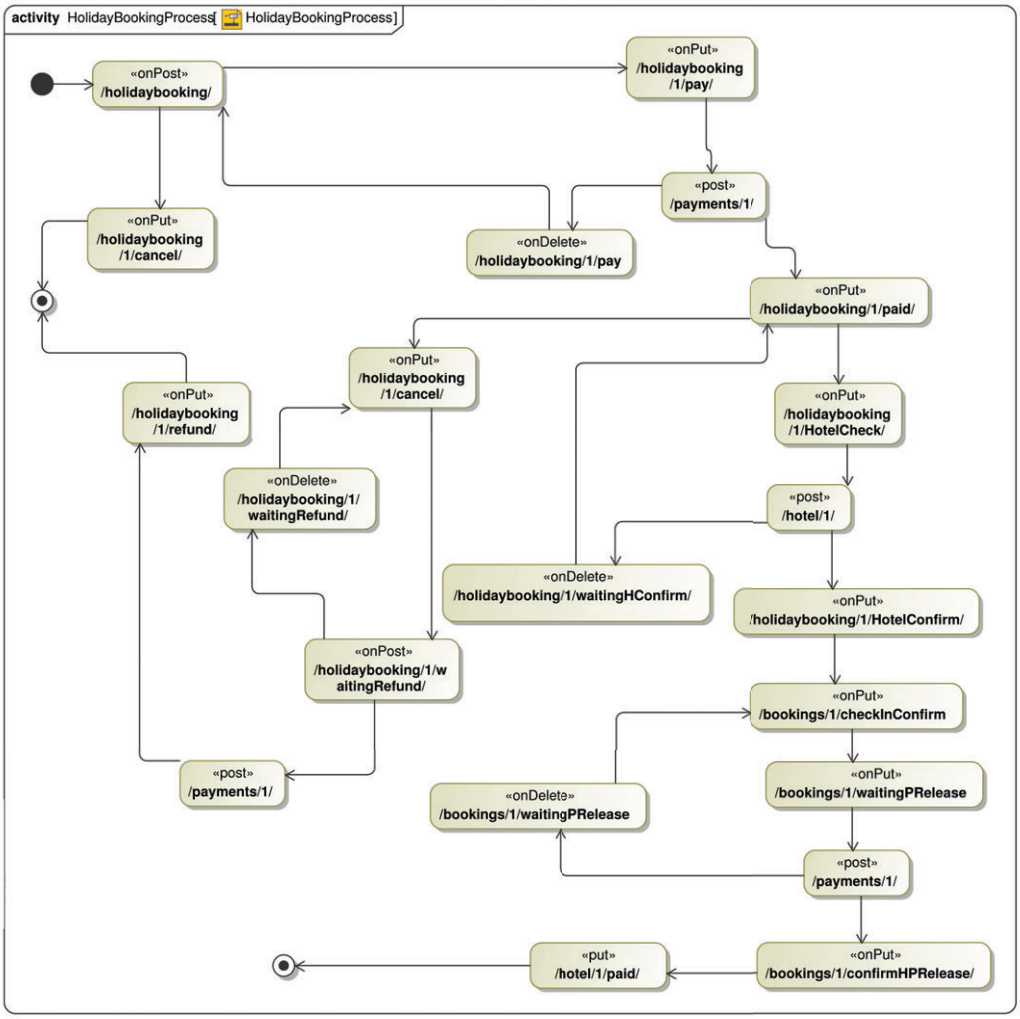


Figure 7.4: Process Model for Holiday Booking REST Composite Service

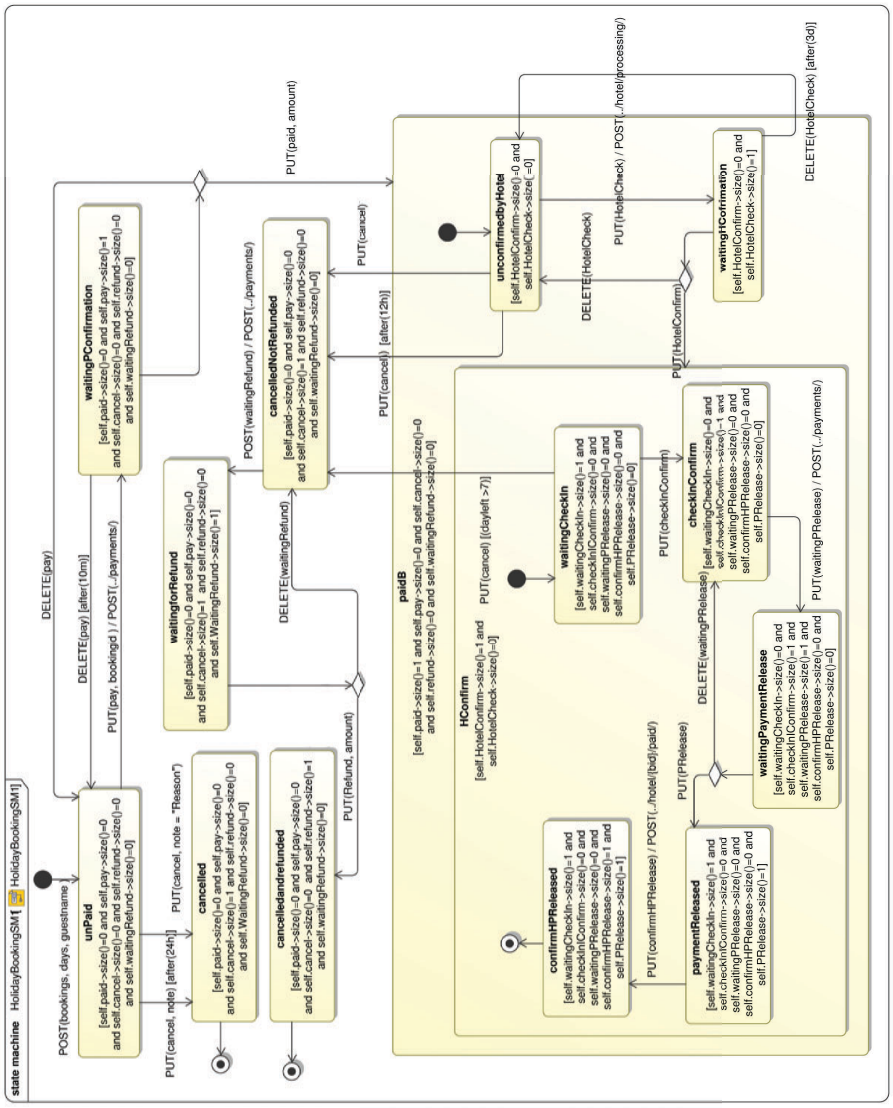


Figure 7.5: Behavioral Diagram of Holiday Booking Composite REST Web Service

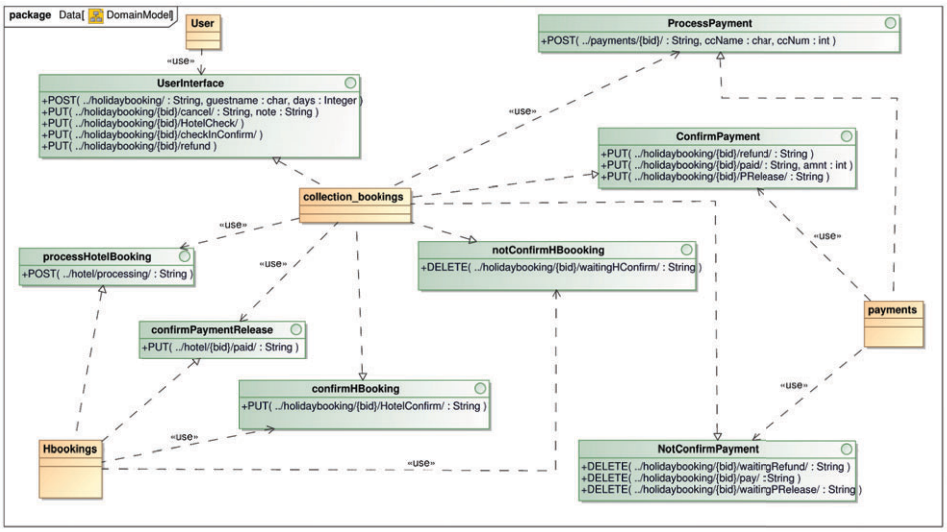


Figure 7.6: Domain Model for Holiday Booking Composite REST Web Service

booking is canceled by the system if it is not paid for 24 hours. A paid booking is checked with the hotel for re-confirmation by invoking a POST on Hotel Service (Figure 7.7 (bottom)). The Hotel Service responds either by invoking a PUT on HotelConfirm or DELETE on HotelCheck making it unconfirmed. An unconfirmed paid booking is then processed for refund by the system in collaboration with the payment service. When a user checks in for a paid confirmed booking, the system invokes the payment service for the release of the payment to the hotel. Once the payment release is confirmed by the payment service, the hotel is notified about it.

The examples provides different interaction scenarios between the composite and partner services while maintaining the state of the composite REST web service. This shows how a stateful REST web service offering a complex behavior can be modeled using our design approach.

7.6 Related Work

In this section, we discuss important related works done by other researchers in the area of specifying and modeling web service compositions. A lot of work has also been done in using UML for web service compositions. Some of these works are also reported in our previous work [109]. Most of the works use or propose a UML profile to support web service composition. Those who do not use UML profile, do not fit our needs since we were interested in defining state of the service using stateless protocol. In the area of RESTful web service compositions, we found the following works noteworthy.

Zhao and Doshi present a formal model for RESTful web services in [132] to

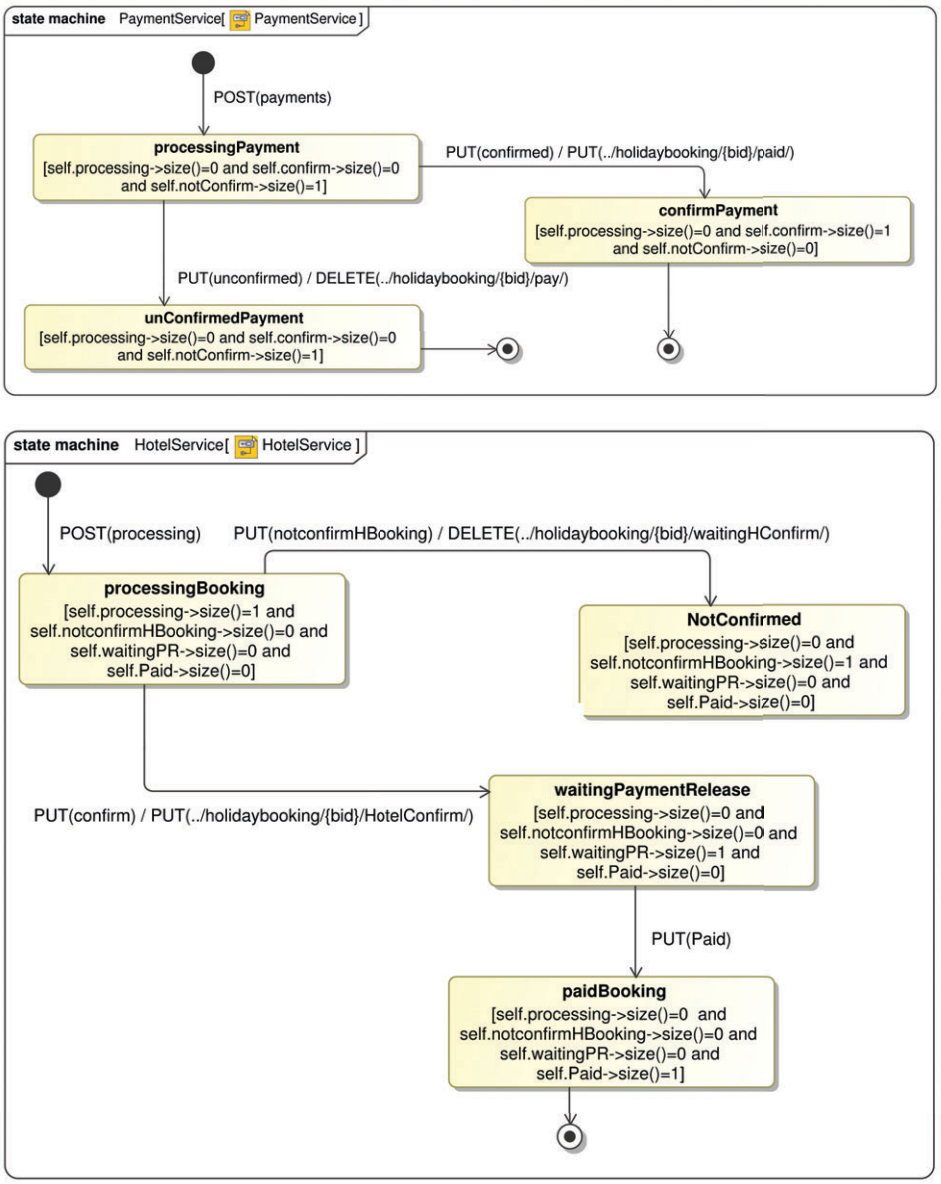


Figure 7.7: Behavioral Model for (Up) Payment REST Web Service (Down) Hotel REST Web Service

automate composition of RESTful web services. Authors define a classification of RESTful web services as Resource Set Service, Individual Resource Service and Transitional Service. These are modeled with ontology concepts and SWRL rules. These models are then used to define state transition system for automating the composition of RESTful web services based on situation calculus. Their work addresses modeling of uniform interface and state transferring between resources. We are however of the view that by defining meaningful URI of the services, the third type 'Transitional Services' may not be necessary.

Pautasso [102] [100] give REST extensions for BPEL that are based on the set of requirements identified for RESTful composition. These extensions aim to support the process of REST composition using the same process-oriented service composition language that supports traditional web services. The work also demonstrates how a RESTful API can be implemented using BPEL with the proposed declarative constructs by exposing selected parts of its execution state. In another work [101], Pautasso identifies set of requirements that should be satisfied by languages for RESTful service composition. A detailed example is presented to understand this set of requirements and implemented with JOpera. While these works complement our work, our focus is on modeling the web service compositions from static and dynamic perspectives. In terms of modeling, Pautasso propose extensions for Business Process Modeling Notation (BPMN) to support REST. The work aims to provide simple extensions for applying BPMN notation to model RESTful business processes. While this works makes good use of BPMN for modeling RESTful business processes by resusing existing graphical elements of BPMN as much as possible, our research aimed to use a modeling notation that is not specific to any domain in particular. BPMN supports concepts of modeling that are applicable to business processes. We are interested in using models that can be used at other phases of developing composite web services and that can facilitate the service developer to verify and validate their designs by using existing mature tools as much as possible.

Rosenberg et al. [112] introduce Bite, that is a lightweight workflow-based composition model for web applications. It aims at simplicity and short development cycle. The basic Bite process is a flat graph having atomic actions and links between them. The flow uses external services in its flow logic and the composition workflow is published as a composed resource. Bite, however, does not support PUT method.

Zhao et al. [133] propose a method for RESTful web service composition based on linear logic. The method finds composition services at both resource and service invocation method levels. The linear logic sequent represent composition requirements and related inference rules are applied to determine if the composition requirements can be achieved.

Yu et al.[130] emphasize on the importance of role in the service description and composition. They present role-centric service descriptions and composition mechanism by proposing a resource meta-model. The RESTful service descriptions

are generated based on the roles that are formally defined on the metamodel. Adopting the BPEL extension for REST approach, authors extend BPEL with the concept of role to have control over the access of resources. Alarcon et al. [13] present a RESTful service composition approach based on Resource Linking Language (ReLL) and Petri Nets. ReLL is presented in [12] as an XML-based description language for REST services with focus on hypermedia characteristics of the model. The work uses a petri net model to illustrate composition model. The approach provides a mechanism that allows to implement a machine client that can perform dynamic discovery of resources. In comparison to this work, our work addresses construction of stateful services with complex scenarios offering all the REST features using models.

More recent work by Bellido [25], analyzes fundamental control-flow patterns in the context of stateless compositions of REST web services. They discuss the challenges of state handling for composed RESTful services in detail and propose means to implement the control flows through callbacks and redirections. The decentralized approach they propose breaks the business process into fragments that represent different intermediary states of the business process and the control logic is centralized in composed resource. The composed resource delegates the control flow to different services and itself becomes available to respond to new messages. When the response is received, services will wake up the composed state at the corresponding state of the execution flow. Compared to their work, our approach targets the state of the services from a different perspective. We define states as predicates over resources. This state information relies on simply invoking GET methods on different resources and formulate the state of the service as a boolean expression based on the response codes of invoked GET methods. In this way, we do not compromise the statelessness feature of REST. This information, however, needs to be stored somewhere. This can either be saved as pre- and post-conditions of methods when implementing the service functionality by the service developer of the composite service or can be implemented as a proxy to provide service monitoring. These service contracts can monitor that the protocol of the service composition is not being violated by any of the parties, i.e. users and service providers. Besides this, our approach provides modeling of the stateful behavior of the REST web service that facilitates the design process of the service. These design models have manyfold applications as discussed earlier and can also be used in verification and validation phases.

7.7 Conclusion

Web services can be composed together to create a new web composite web service. This composite web service combines functionality of its partner services with a business goal in mind such that the functionality of new composite service is an aggregate of its partner services. In this chapter we showed how REST web

services can be composed to create a composite web service with REST features. In doing so, we have extended our modeling approach to support RESTful web service compositions. The static structure of the composition is modeled as a resource model and the behavior of composite REST web service is modeled with the sequence diagram, activity diagram and state machine diagrams. The sequence diagram models the process scenarios, the activity diagram shows the process flow and the behavioral model represents the behavioral interface of composite REST web service. We also show mapping mechanism from activity diagram to BPEL-REST. In addition, the service design models can be directly mapped to implementation languages like Django web framework, Ruby on Rails and UPPAAL timed automata etc.

We have applied our approach on a relatively complex worked example of a holiday booking web service available on the Internet. The example shows how a REST composite web service with complex behavior can be constructed following our design approach.

Chapter 8

Validation of Services

The use of REST web services in businesses and critical applications motivates the need for validation approaches that would effectively and efficiently detect faults in their specifications and implementations.

We have earlier explained our design methodology to create behavioral interfaces for simple and composite web services (CWS) that are REST compliant and offer complex scenarios and timed behavior. These service design models and their implementations should be validated for their correct behavior in order to build trust on the service functionality. In this chapter we present the validation approach that facilitates the service developer in creating dependable REST web services. We have used model checking approach for this purpose. Model checking is a way to exhaustively and automatically check if a finite-state model of a program satisfies its specifications [46]. The goal is to see whether the models have basic properties like deadlock freedom and other critical properties the absence of which can cause a system to crash.

In our approach, a service can invoke other services and exhibit complex and timed behavior while still complying with the REST architectural style. We need to check if the service implementation is functioning correctly alongwith partner services and if the service goals and timed constraints are being fulfilled. We, thus, show how we validate the implementation of the RESTful web service composition with model-based testing approach. By using model-based testing (MBT) approach, automatic test cases can be generated with an increased probability of test coverage and with an ease of test case maintenance.

In section 8.1, we present our validation approach explaining different steps of validation approach in detail. The transformation steps from UML to UPPAAL Timed Automata (UPTA) are presented in section 8.2. The approach is applied on a Holiday Booking RESTful web service in section 8.3 and the validation of the approach is given ins secion 8.4. In section 8.5, we present our work on contract based testing to validate classes and web services. The related work from literature is presented in section 8.6 and section 8.7 concludes the chapter.

8.1 Validation Approach

Our validation approach takes as input the behavioral interface specifications (design models) of the composite REST web service, its partner REST web services and the business requirements. We then provide verification of the design models by reasoning on the basic properties of models like deadlock, liveness, reachability and safety with UPPAAL model checker. UPPAAL is a commonly used model checking tool for verifying real time systems through modeling and simulation [82]. However, the UML-based service design models represent the system graphically and are comprehensible for a human user. In order to make the models amenable for model checking tools we suggest a set of reversible mechanized steps for translating UML-based service specifications into UPPAAL timed automata (UPTA) [81]. UPTA are then simulated and verified with UPPAAL model checker. UPTA are updated (if needed) based on the verification results and transformed back to UML.

From the UML models, a skeleton of the composite service is generated automatically in Django web development framework [66] using our partial code generation tool presented in Chapter 9. Performing the verification of the web service composition in a model-checking tool allows us the increase the quality of the specifications before proceeding to the implementation of the service.

The transformation step from UML to UPTA consist of generation of two types of automaton from service design models. One automaton corresponds to our service design models and the other represents the environment model. The environment model simulate the behavior of service user to invoke the interface service methods in order to facilitate test generation automatically.

For model based testing of the service implementation, we have used online conformance testing tool UPPAl-TRON that validates the service implementation against its UPTA specification models at runtime. For this purpose, we have customized and modified the Tron test adapter to establish connection between UPTA models and implementation under test. In the end, we have evaluated the validation approach for its efficiency using mutation testing and other different coverage and benchmarking tools.

Requirements traceability is also an important component of our integrated approach. The requirements of the composition are included in the UML specifications and then propagated to UPTA specifications. They are used for both verifying the reachability of those model elements implementing them and for reasoning about their coverage after the tests are executed. Upon detecting failures, traced requirements can be used to localize the errors in either models or in the specifications.

More details of the approach are given in the following:

8.1.1 Verification

Model verification is a process of determining whether the models are designed correctly and represent the developer's conceptual descriptions and specifications. Model checking is one of the ways to exhaustively check the models automatically. The service design models of composite REST web service should be verified for their basic properties in order to build confidence of the service designer on the models before implementing them. This allows one to eliminate design errors that can be expensive to detect and correct at later stage of the development cycle.

UPPAAL model-checker is used for modeling, simulation and verification of real-time systems [82]. It consists of set of timed automata (TA), clocks, channels that synchronize the systems (automata), variables and additional elements. A real-time system is modeled as a closed network of TA. Each automaton in the network is specified via a *template*, which can be instantiated as *process*. A template in UPTA is composed of *locations*, *edges*, *clocks* and *variables* representing all properties of the system. Synchronization between different processes can be provided using *channels*. Two edges in different automata can synchronize if one is emitting (denoted as *channel_name!*) and the other is receiving (denoted as *channel_name?*) on the same channel. *Guards* are the conditions that enable a transition when they are satisfied. Similarly, the conditions associated to locations, called *invariant*, specify that the system can stay in the location if and only if the invariant is satisfiable.

The query language used in UPPAAL is a simplified version of TCTL [15] that consists of state formulae and path formulae. State formulae (φ) is an expression that describes an individual state, while path formulae can be classified into reachability, safety and liveness properties. Deadlock is expressed using state formulae. The syntax of TCTL path formulae that are used in UPPAAL is defined as follows:

- A $\Box$ φ - for all paths, the property φ is always satisfiable.
- A $\Diamond$ φ- for all paths, the property φ is eventually satisfiable.
- E $\Box$ φ - there is at least a path in the automata such that property φ is always satisfiable.
- E$\Diamond$ φ - there is at least a path in the automata such that property φ is eventually satisfiable.
- $\varphi \rightsquigarrow \phi$ - when φ holds, ϕ must hold.

If there is a location in the model that has no outgoing transition, then the model is said to be in a deadlock. Reachability properties validate the basic behavior of the model by checking whether a certain property is possible in the model with the given paths. The safety property checks that something bad will never happen and the liveness property determines that something will eventually happen.

However, before using UPPAAL model-checker to verify these properties, we need to give service design models formal foundations that are understandable by the verification tool. This has to be done in an automated manner to avoid extra efforts from the service developer. In section 8.2, we present our tool support for UML to UPTA transformation.

8.1.2 Code generation

We can generate the code-skeleton of service design models using our tool presented in Chapter 9. The tool generates code skeleton for design models in Django that is a high level Python web framework [66]. The generated code also has behavioral information, i.e., the pre- and post-conditions for each method and the developer has to only write the implementations of the operations.

8.1.3 Requirements Traceability

Service requirements can be inferred from the specification document and they serve as service goals. A service should be checked for its service goals in order to validate that the service does what it is required to do. By catering to the service requirements at the design phase and propagating them to the validation stage, we provide a mechanism by which a service requirement can be validated for its goals and the unfulfilled requirements can be traced back to the design phase to find faults in the design. Service requirements are generally domain-specific since they are inferred from the specifications. We infer functional and temporal requirements from the specification document into a table and number them. These requirements are attached to the behavioral model as *comments* on the transitions and are propagated to UPTA such that the links between requirements and the model elements are preserved. These requirements are included in all the models and traced throughout the process, i.e., at UML, UPTA and test level, respectively.

The requirements are formulated as reachability properties in UPTA with the purpose of verifying them during simulation. Each requirement label is translated into a boolean variable (initialized to False) and attached to the corresponding edge in the UPTA. This mapping is explained in more detail in the Section 8.2 on the UML to UPTA transformation.

We require that our testing approach must validate that these requirements are met by IUT, in order to build confidence of the developer that the system is doing what it is required to do. Thus, their coverage level is monitored during test generation and execution. Once the test report is available, we can check which requirements have been validated and which have failed.

8.1.4 Model-Based Test Generation

Model-based testing (MBT) is a method that provides an abstract model of a system under test (SUT) and preforms automatic test case generation based on the specifications of the SUT [126]. In MBT, modeling the environment of a system is important since the environment generates test cases from whole or some parts of the model to satisfy the test criteria. Environment models help in automation of testing in three ways: the automation of test case generation from a simulated environment, the selection of test cases, and the evaluation of their test results. Our

UML to UPTA transformation tool generates both the behavioral model of SUT and the environment model.

We provide automatic test generation using UPPAAL TRON, which is an extension of UPPAAL for online model-based black-box conformance testing [81]. During test generation, the environment model randomly selects test cases and communicates to the test adapter.

A test adapter is used by UPPAAL TRON to expose the observable I/O communication between the test environment model and the SUT model, as shown in Figure 8.1. Our adapter implements the communication with the SUT by converting abstract test inputs into HTTP request messages and HTTP response messages into abstract test outputs. The TRON tool generates tests via symbolic execution of the specifications using randomized choice of inputs. Based on the timed sequence of input actions from the simulation, the adapter preforms input actions to Implementation Under Test (IUT) and waits for the response. Output from IUT is monitored and generated as output actions for the simulation. The conformance testing is achieved by comparing outputs of IUT to the behavior of the simulation.

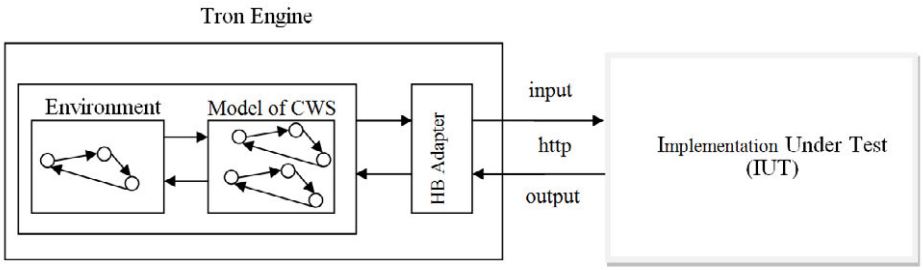


Figure 8.1: UPPAAL TRON test setup

8.2 Design Models → UPTA transformation

The transformation from service design models to UPTA consist of series of steps presented in this section. We require to generate two UPTA for our behavioral model: one UPTA corresponding to the behavioral model of the service and one environment model that simulates the user behavior and triggers the UPTA of the service.

A real-time system is modeled as a closed network of timed automata, i.e, UPTA. Each UPTA in the network is specified via a template. These template can be instantiated as processes. UPTA is composed of locations, edges, clocks and variables representing different properties of the system. We translate our resource, behavioral and domain models to UPPAAL in order to verify them with UPPAAL model-checker.

8.2.1 Resource Model

In UPTA the resource model is represented as a template. The *resource definitions* in the resource model are specified as variables with 1 or 0 value, specifying if a resource exists or not, respectively. The attributes of *resource definitions* are inspected and for each integer attribute, an integer variable is declared in the UPPAAL model. Similarly, the boolean attributes are declared as integer arrays of 0 and 1.

8.2.2 Domain Model

The domain model shows set of operations offered and required by the composite web service and its partner web services. The corresponding communication between templates in UPPAAL is represented by channel synchronizations. Two edges in different automata in UPPAAL can synchronize if one is emitting and the other is receiving on the same channel. The operations in an interface are thus translated into a binary synchronization channel in UPPAAL. The template of the service that realizes the interfaces acts as the receiving automaton and the sending automaton is specified by the template of the service that uses the interface.

8.2.3 Behavioral Model

The behavioral model of REST web service is encoded by timed automaton that are represented by templates, which are instantiated as processes. Figure 8.2 shows an example of transformation from the behavioral model to UPTA.

States

A state is mapped to a location in UPTA, and a state invariant is mapped to corresponding location invariant. The subclauses of the state invariant are translated to variables corresponding to the respective resource definition. For example, in Figure 8.2, $self.a->size() = 1$ is translated as *a = 1* and $self.b->size() = 0$ as *b = 0*. The initial state corresponds to the initial location. The final states are translated by having an edge from the corresponding location to initial location and updating all the variables to their initial values, as shown in Figure 8.2. The choice state in the behavioral model is replaced by two edges in the TA model that are originating from the same source location to different target locations.

State Hierarchy

The behavioral model may contain composite states for better representation of specifications. UPTA, however, does not support the notion of location hierarchy. We flatten the composite states to several simple states by including the state invariant of super states in the contained states that are then mapped to the respective

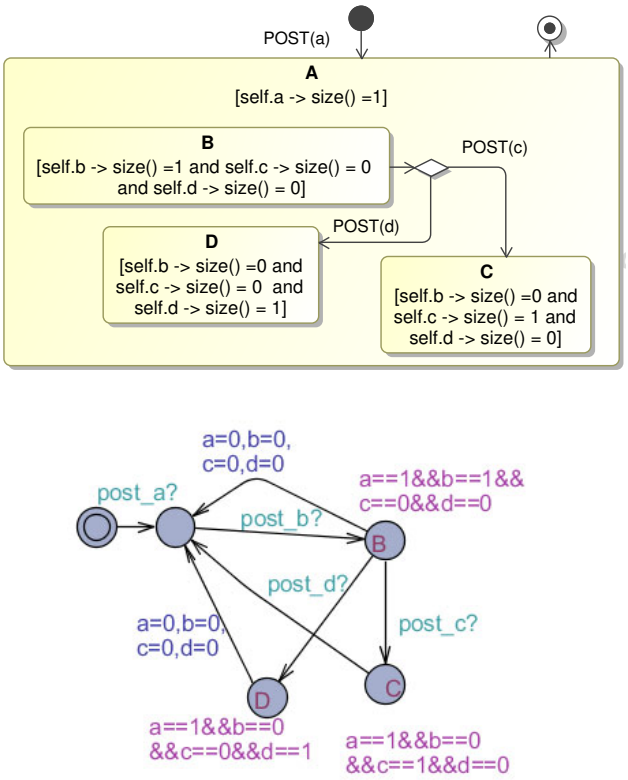


Figure 8.2: (Top) Composite State in Behavioral Model. (Bottom) Flattened locations in UPTA

locations in UPTA. For example, in Figure 8.2, the top figure contains a behavioral model with a composite state that is flattened to UPTA model shown at the bottom. States B, C and D in the behavioral model correspond to the locations B, C and D of UPTA, respectively. Note that all the locations contain the state invariant of superstate A in the behavioral model.

Transitions

A transition in the behavioral model is mapped to an edge in UPTA and guards on the transition are mapped to guards on the corresponding edge in UPTA. In Figure 8.3, we show how the transitions in behavioral model (top) are translated to UPTA (bottom right). The locations $L1$ and $L5$ correspond to states $S1$ and $S2$, respectively, and locations $L0$, $L2$, $L3$, and $L4$ are the extra locations created during the transformation process as explained below. The state invariants are translated to location invariants and represented as boolean functions for the purpose of diagram clarity. The transition between states S1 and S2 is triggered by POST(b) after 10 minutes. In UPTA, this is represented as guard over the clock variable *cl*.

Trigger Methods

The trigger methods from the behavioral model are translated in to receiving channels in UPTA. This receiving channel is in sync either with the automaton of the partner service or with the environment model.

Time Events

The time events in behavioral diagram are replaced by clocks in UPTA. The clock is reset in the incoming edge to the location (L1) and is also included in the location invariant. Thus, the guard *after(10m)* is translated to $cl > 10$ on the corresponding UPTA edge.

Effects

The effect on the transition, i.e., $POST(c)$ shows invocation to the partner service. The communication between two web services is established by using a unique channel synchronization. For instance, emitting a request from a web service to the other one can be replaced by synchronizing a channel in an UPPAAL process, and the other process is the receiver of the synchronization. The effect of the transition that invokes a remote service is represented with two edges and an urgent location (marked with U in the circle) in between, i.e., edges *e2* and *e3* and *urgent* location *L3*. An urgent location in UPTA does not allow any delays [82]. Thus, the first edge (e1) is synchronized with the environment model and the second edge (e2) synchronizes with the partner automaton. The third edge (e3) is synchronized to receive acknowledgment response from the partner (as we model asynchronous service) and the sending channel on the fourth edge (e4) is synchronized with the environment to indicate the completion of transition.

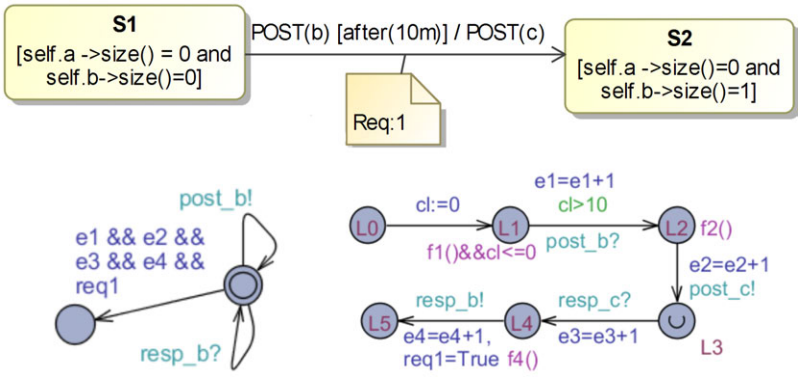


Figure 8.3: Example of behavioral model (top) Corresponding Environment Model (bottom left) and Flattened UPTA (bottom right)

Requirements

The requirement on the transitions are translated into a boolean variable (initialized to *False*) and attached to the corresponding edge which updates it to *True*. This is shown in Figure 8.3 with *Req1= True* on edge e4. This implies that whenever this edge would be traversed, this requirement will be met. This can be formulated as reachability properties to attain requirement coverage and tracked during test generation and execution.

8.2.4 Environment Model

The environment model in UPTA has sending channels that are received by the composite web service automaton as inputs to trigger the process. This is similar to interface method calls in the SM. All the interface methods of the service specified in the state machine are mapped to the sending channels in the environment model and the response of successful transition is received from the composite web service via receiving channels. This is also shown in Figure 8.3: the environment model initiates the automaton (bottom right) by sending channel *post_b*! and the process completes when the channel *resp_b*? is received.

A Python script is currently used to create the environment model, from a given UPTA model by analyzing the channels observable from the environment. The original idea has been discussed in [65]. This will be merged in the final version of the UML→UPTA transformation script.

8.2.5 Test Coverage information

In order to enable rigorous test coverage in UPPAAL TRON, a second Python script (discussed in more detail in [75]) is used to automatically add *counter* variables for each edge of a given automaton in a UPTA model and a corresponding update of the given variable on the corresponding edge. Whenever the edge is visited during the simulation or execution, the variable is incremented, allowing thus to track which edges have been visited and how many times. This enables one to track coverage level wrt. e.g., edge coverage or edge pair coverage. This script will also be integrated in the final version of the UML→UPTA transformation script.

8.3 Case Study

We have demonstrated our approach with a worked example of a Holiday Booking (HB) REST composite web service (CWS) that is built on inspiration from the *housetrip.com* service. It is a holiday rental online booking site, where you can search and book an apartment in the country of your destination. The modeling of this example service is explained in the previous chapter. In this chapter, we are interested in validating the service design models and its implementation. The

example provides a good demonstration of how a composite REST service with complex stateful behavior is designed and validated using our approach.

8.3.1 Design Models

The design models are modeled using MagicDraw [6]. Static validation of models is done via OCL using the validation engine of Magic Draw. We rely on predefined validation suites for UML contained in MagicDraw for the basic validation of the model. These validation suites contain rules that check that the designed UML model conforms to UML metamodel specifications and prevent the developer from creating incorrect models.

The composite REST web service and its partner services are modeled with resource, behavioral and domain models. These models for holiday booking REST composite web service are shown in Figure 8.4, Figure 8.5 and Figure 8.6, respectively. Figure 8.7 and Figure 8.8 show the behavioral model of partner services, i.e., payment Service and hotel service, respectively. The construction of these models has been explained in the previous chapter.

8.3.2 Verification

The design models of holiday booking composite REST web service are translated to UPTA with the help of tranformation tool. Figure 8.11 shows UPTA of holiday booking composite REST web service. Figure 8.9 and Figure 8.10 show the UPTA of partner services, i.e., payment Service and hotel service, respectively.

The verification properties are specialized for our case study and some of them are mentioned below.

Deadlock Freeness. The holiday booking service, the hotel service and the payment service models are all deadlock free. This means that the composite service is never reach to a state that cannot preform a transition (i.e., $A[]\ not\ deadlock$).

Reachability. All the locations in the HB service are reachable. This means that the model receives and sends messages to the partner services smoothly and the model is validated for its basic behavior (i.e., $E \diamond CompService.r$), where r is the last location in the TA model and indicates that all processes for certain booking is completed.

Safety. Some of the safety properties in our model are: a) Payment should be released iff the user has checked in, i.e., ($E \Box CompService.h2$ imply $CardService.c2$), where *c2* is the location after check-in and *h2* is the location after payment release, b) If the payment is released by the HB service then the Hotel service is paid, i.e., ($E \ \Box\ CompService.h2$ imply $HotelService.p$), where p is the location in Hotel service model for hotel payment.

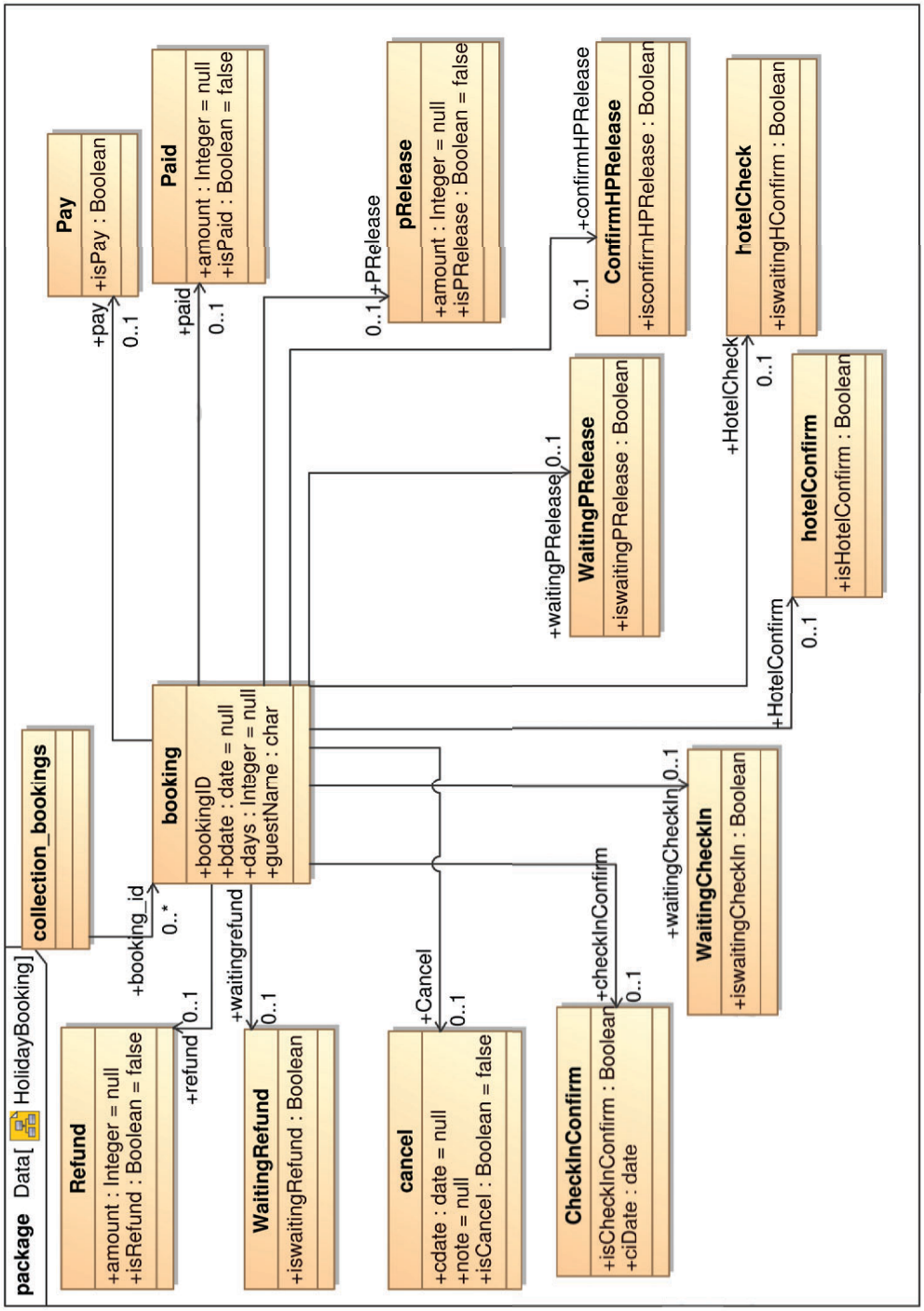


Figure 8.4: Resource Diagram of Holiday Booking Composite REST Web Service

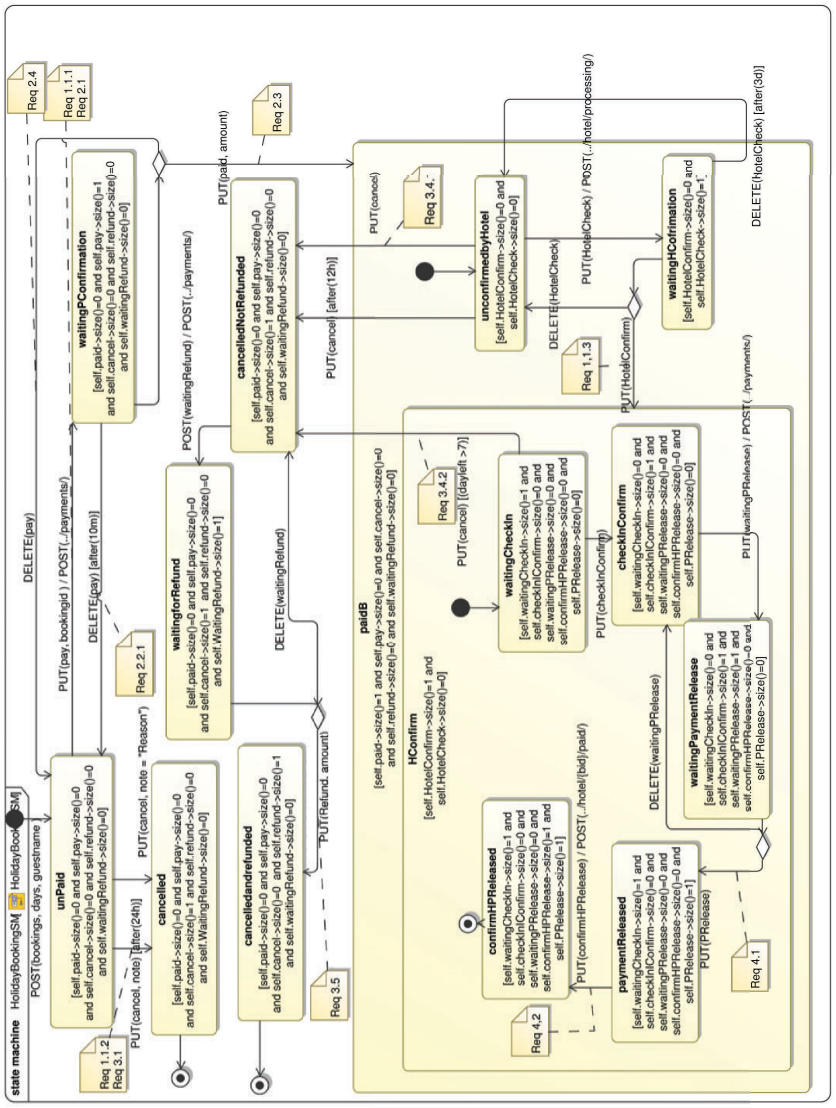


Figure 8.5: Behavioral Diagram of Holiday Booking Composite REST Web Service

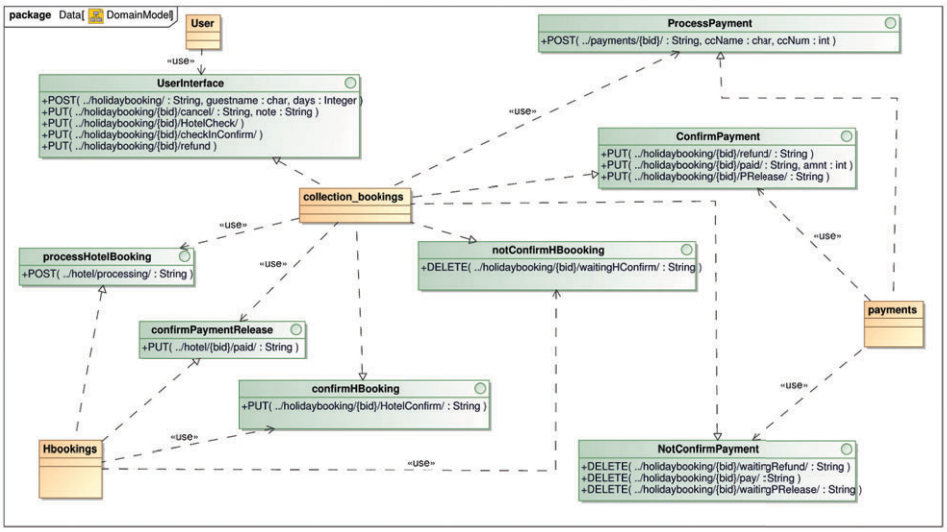


Figure 8.6: Domain Model for Holiday Booking REST composite web service

Liveness. Some of the liveness properties in the model are: a) When the payment is not paid within 24 hours, the booking is canceled (i.e., *CompService c* and *compService cl* > 24 *CompService b*1), where *c* indicates waiting for the payment, *cl* indicates clock of the model and *b1* indicates the booking request is going to cancel due to the delay, b) If the Hotel Service does not confirm within 3 days then the booking is considered not confirmed (i.e., *CompService o* and *CompService cl* > 3 *CompService n*), where *o* is the location for waiting for the hotel response and *n* is the location for canceling.

8.3.3 Requirements Traceability

We have inferred functional and temporal requirements from the specification document. Table 8.1 shows the requirements for holiday booking RESTful composite web service. These requirements should be fulfilled by the IUT in order to satisfy the service goals. These requirements are added as comments on Figure 8.5 and translated to a boolean variable (initialized to *False*). These variables are attached to the corresponding edges in the UPTA assigning a *True* value. When the corresponding edge is traversed, its value becomes *True*. We attain requirement coverage by encoding them as guards to edges in the environment model. Whenever all the requirements evaluate to TRUE, the environment model can go to the final location. These values are encoded as boolean function `verdict()` in Figure 8.12.

Test setup. The testing process includes the TRON engine, an adapter, the IUT and the model of system. The TRON engine establishes TCP/IP connection on a local port and via that the adapter starts communications. The adapter works

Table 8.1: Requirements Table of Holiday Booking REST CWS

Req	Sub-Requirements
1- Booking	1.1 - A booking should be paid 1.1.1 - A booking should be paid within 24 hours of the booking. 1.1.2 - If a booking is not paid within 24 hours of the booking, then it is canceled by the system 1.1.3 - A confirmed paid booking, waits for user check in
2- Payment	2.1 - When user pays for the booking, partner service should be invoked to process the payment. 2.2 - The HB CWS should wait for response from the payment service 2.2.1 -If the payment service does not respond in 10 minutes, it is considered not working and the booking is marked unpaid 2.3 - If the partner service confirms the payment, the booking should be marked paid 2.4 - If the partner service unconfirms the payment, then the booking should be considered unpaid.
3- Cancel	3.1 - A booking is canceled if it is not paid for 24 hours 3.2 - A paid booking that is not confirmed by the hotel is marked unconfirmed 3.3 - A paid booking that is unconfirmed by the hotel is canceled after 12 hours. 3.4 - A paid booking can be canceled by the user 3.4.1 - A paid booking can be canceled by the user if it is not waiting for payment confirmation or hotel confirmation. 3.4.2 - User can cancel paid booking only before 7 days of checkin. 3.5 . A canceled booking must be refunded.
4- Payment Release	4.1 - If the user checks in then the payment must be released to the hotel. 4.2 - When the payment is released to the hotel, HB CWS must notify the hotel about release of the payment

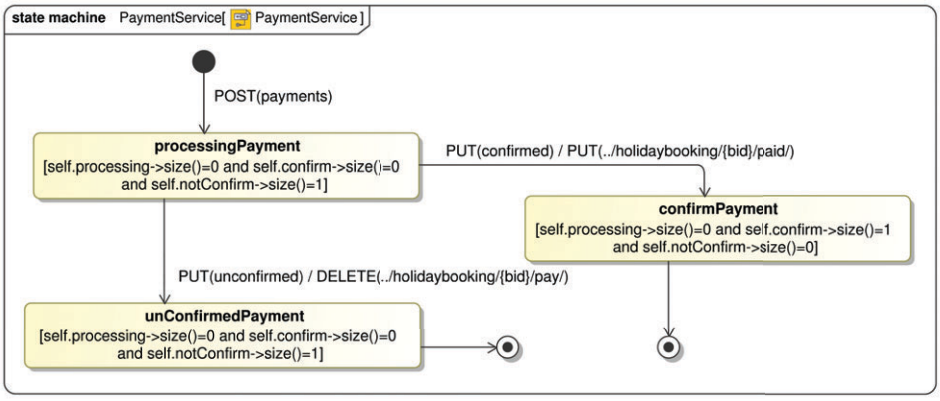


Figure 8.7: Behavioral Model for Payment Service

as an interface, which translates the inputs and outputs of the model to proper HTTP requests to/from the IUT. The UML to UPTA transformation also generates (optionally) a skeleton of the test adapter, depicting what interfaces should be implemented between TRON and IUT. Once the adapter is fully implemented, it can be reused for different versions of the models as long as there is no new I/O messaging being done. The IUT is a web service composition of three web services: HolidayBooking, Hotel and Payment Services.

Composition Model. We specified a TRON adapter which communicates between IUT and the composition model an excerpt of which is shown in Figure 8.11. The adapter identifies the emitting and receiving channels as well as defines the corresponding HTTP request functions. During test execution, the composition model waits until a channel call comes from the environment model, then the adapter translates the incoming channel to a specific HTTP request and sends it to IUT. The response from IUT will be checked in the adapter and forwarded to the composition model as a response channel.

8.4 Validation of Approach

Our holiday booking composite REST web service had 14 states and 25 transitions. These were translated into a UPTA model with 34 locations and 46 edges. Similarly, the state machines of Payment service had 3 states and 4 transitions which transformed in to a UPTA model with 5 locations and 6 edges. The Hotel service had 4 states and 5 transitions that were translated into 7 locations and 9 edges. In addition, the environment model created had 4 locations and 13 edges.

One issue with using formal tools like UPPAAL for verification and test generation is the scalability of the approach due to the state space explosion. In

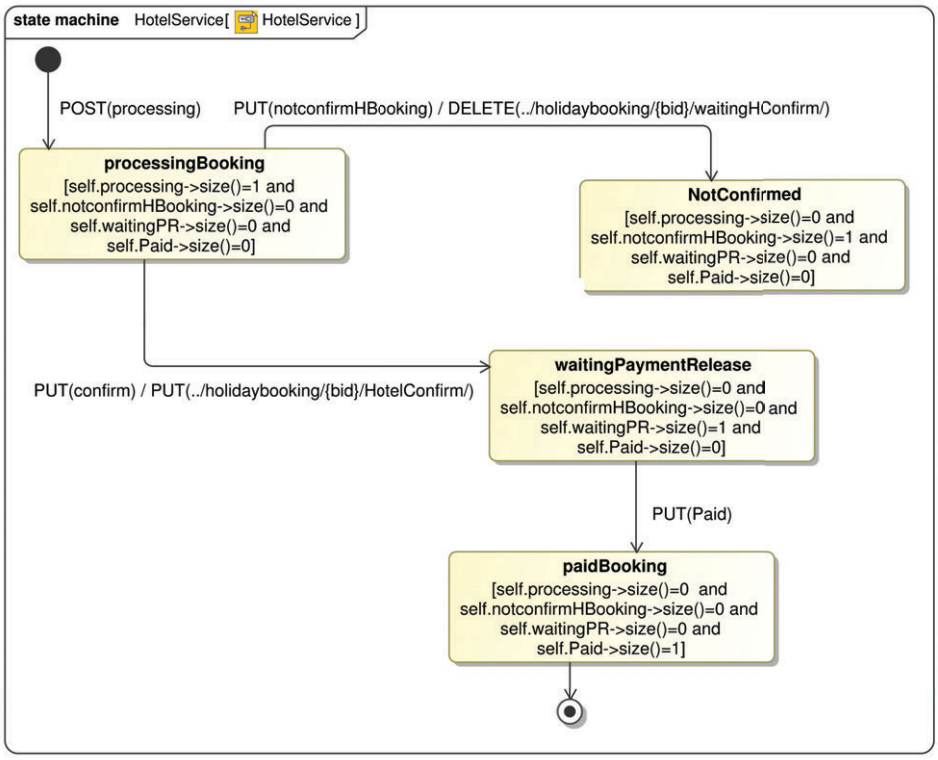


Figure 8.8: Behavioral Model for Hotel Service

contrast to offline test generation, where the entire state space has to be computed, in online test generation only the symbolic states following the current symbolic states have to be computed. This reduces drastically the number of symbolic states making the test generation less prone to space explosion and thus more scalable. For instance, the number of explored symbolic states when generating with the `verifyta` tool, the traces satisfying complete edge coverage (i.e., $\&e_1 \quad \&\&e_m$, where e_j are tracking variables corresponding to all m edges of the models) was 974. In the contrast, the maximum number of symbolic states reported by TRON during a test session achieving complete edge coverage was 12 (see Figure 8.13).

Figure 8.13 plots the evolution of the number of symbolic states for 10000 model time units (10 seconds). The number of states in specific testing time depends on the behavior of the model on that time. For example, from the time 100 to 160, the payment process is running. Due to the timing constraints of the model, it is not obvious that which response will return and hence the next location in the model is not determined. Therefore, the number of the symbolic states are more. At time 190, the booking web service communicates with other web services (Hotel-service and Card service), and the total number of symbolic states is the highest (12 states).

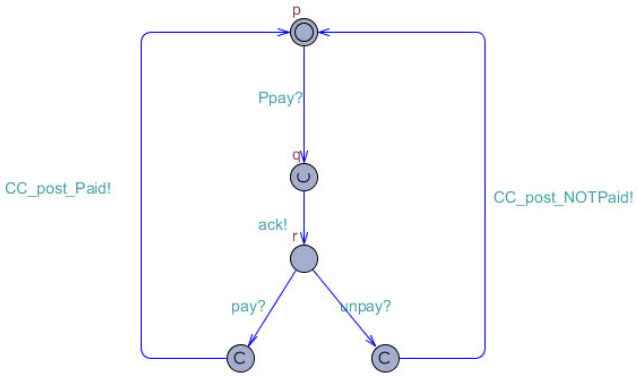


Figure 8.9: UPTA for Payment Service

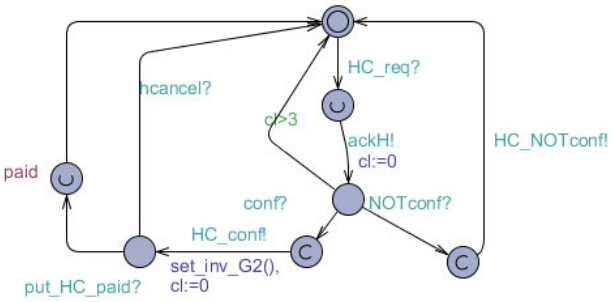


Figure 8.10: UPTA for Hotel Service

For benchmarking the verification process, we have used the `verifyta` command line utility of UPPAAL for verification of the specified 5 properties. We have used the `memtime` tool to measure the time and memory needed for verification. The result showed in average 0.20 seconds and 54996 KB of memory being used. Although the memory utilization depends heavily on the symbolic state space, it shows that the current size models leave room for scalability of the approach. A known limitation of UPPAAL is that the maximum memory size it can use is close to 4GB due to its 32-bit architecture.

In order to evaluate the efficiency of our approach, we compared the specification coverage with the code coverage yielded by a given test run. Since we had access to the source code of the IUT, we used the *coverage* tool for Python [1] to report the code coverage for each test session. The Table 8.2 lists results of several measurements.

The complexity of the models resulted for the holiday booking service in Figure 8.11, allowed us to verify all specified properties in UPPAAL and to generate tests

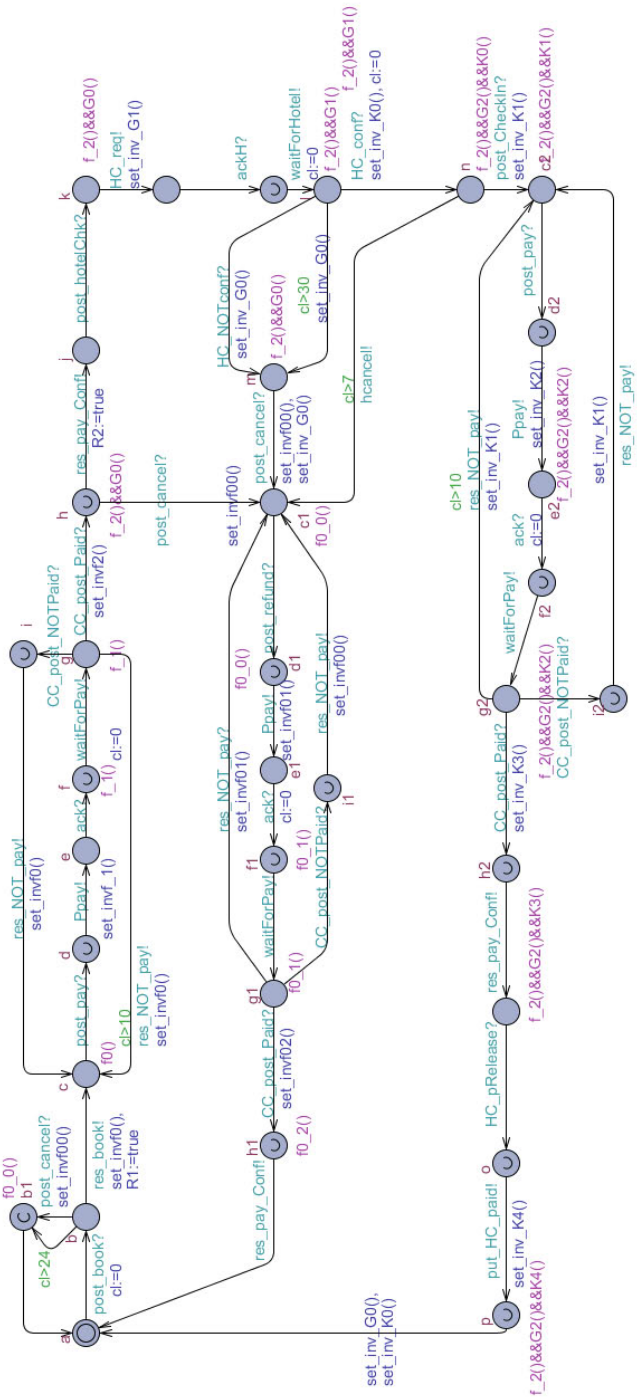


Figure 8.11: UPPAAL automata of Holiday Booking Composite REST Web Service

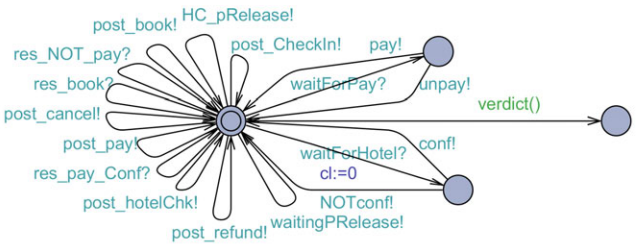


Figure 8.12: Environment model

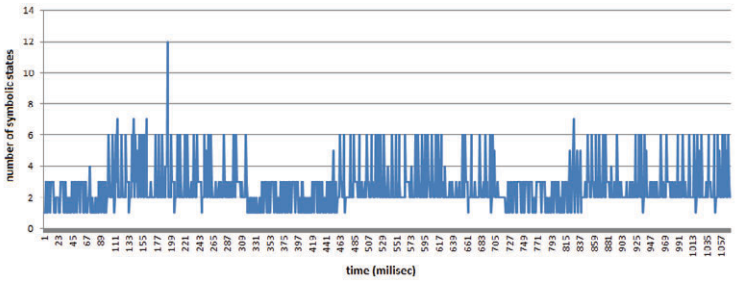


Figure 8.13: Evolution of symbolic states

using TRON. To overcome the state explosion problem, the models would either be optimized at UPPAAL level, or they can be split into several parts, via slicing or aspect-oriented approaches, each focusing on a different concern of the system.

Table 8.2: Correspondence between code coverage and edge coverage

Run	Edge Coverage	Code Coverage
1	64 %	55%
2	80%	67%
3	100%	78%

Although many of the errors were caused by modeling mistakes, testing revealed some errors in the implementation as well. For instance, in the holiday booking service, there was an error in sending *cancel* request and another error was found in the POST header in *refund* request. In the hotel service, confirmation was sent by a wrong method, so it was rejected by holiday booking service.

In order to evaluate the fault detection capabilities of our approach, we have manually created 30 mutated versions of the original holiday booking service program code. Each mutation had one fault seeded in the code, for instance replacing POST with DELETE, removing one line of the source code, change of

logical conditions, etc. The faults were always seeded in those parts of the code that is covered when achieving 100% edge coverage of the model. We assumed that the original version of the program is the correct one. We assumed that the original version of the composite web service is the correct one, as we were able to run the 100 test sessions in TRON against it. For each mutated version of the composite web service, we set the TRON to execute 100 test sessions against it. When a fault was discovered, down and the mutant was considered as *killed*. If the mutated statement has been covered by the test runs but no failure was detected, we mark it as *alive*. Out of the 30 mutated programs, 28 mutants were killed and 2 are alive mutants. It resulted into a mutation score of 93.3%.

Out of the 30 mutated programs, 28 mutants were killed and 2 are alive mutants. The mutation score calculated based on the following formula:

$$M_s = \frac{N_k}{N_G} * 100\%$$

where M_s, N_G and N_k are the mutation score, the number of generated mutants and the number of killed mutants respectively. Mutation score in this approach was 93.3%. One of the alive mutants was for the time variable that should be set from client side, but since we did not implement client side, the mutant from the time variable does not have effect on the behavior of the IUT. The other alive mutant could be found by the IUT but since it did not change the expected behavior of IUT, the test adopter did not discover it. Overall, the mutation score indicates that the model based testing approach was quite accurate and could cover all expected behavior of the IUT.

8.5 Testing Classes against their Contracts

Service implementations contain information about method contracts derived from the models. We can also take advantage of contract-based testing approaches using service method contracts asserted in the code. The derivation of contracts from a UML protocol state machine has been discussed earlier in section 3.1. By using UML protocol SM to define behavioral interfaces of REST web service and the approach described below to generate class contracts, we can benefit from previous and future efforts in test case generation from behavioral contracts while using a familiar and standardized visual notation.

Class contracts can be used to generate run-time assertions that reveal if a particular execution of the system breaks the precondition or postcondition of a method and to generate test cases to exercise the method's assertions [91, 85, 35]. In this section, we present the research tool that we have developed to automatically extract class contracts from UML protocol state machines according to definitions presented in section 3.1. We then use the asserted class contracts with different testing tools to validate a class.

For java based web services, contract is based in JML [83]. The tool accepts as input a UML 2.0 model serialized as XMI and a Java file containing an interface or a class. Then it automatically updates the Java file by inserting a contract derived from the UML protocol state machine. The tool can be used either as a stand-alone command line python program [7] or as a plugin for MagicDraw UML.

To represent the UML protocol state machine, XML Metadata Interchange is used and the specification is saved into a file with source code corresponding to the specification. The XMI data for this version can be generated from different tools e.g., Papyrus UML, Borland Together and MagicDraw UML. We have created our examples with MagicDraw UML, since this software seems more suitable for this purpose compared to other UML editors.

The generated contracts can be used with other tools to test and validate a Java class that implements the protocol described in the UML protocol state machine. Examples of such tools are the JML JUnit [43] tool that simplifies the development of white-box unit tests, JET [42] i.e., an automatic random test generator for JML, and ESC/Java2 [54] i.e., a static analysis tool.

A typical development task involves the use of a source code editor, a UML modeling tool, the contract generation tool and different testing tools like JML-JUnit, ESC/Java2 and LIME tools. To increase the usability, we integrated most of these tools behind a single user interface. These tools were integrated as a MagicDraw plugin as it was being used primarily for UML modeling. Figure 8.14 shows a screenshot of MagicDraw tool plugin.

We also added a validation feature to the plugin to check the UML protocol state machine before the generation of specifications. It has a built-in component that parses the model and reports the short-comings in the model. The short-comings are the lack of elements or use of unsupported features that results in an incorrect specification. In case the validation is not successful, a user friendly information on faulty elements is displayed to the user in a diagram window. This feature further adds to the usability of the tool. Figure 8.14 shows an example of validation feature.

Statement coverage for JML-JUnit can be calculated during execution of the tests. This is done using JVMDI Code Coverage Analyzer which is an open source shared library loaded into a Java virtual machine (Java 1.4 or 1.5). The program reads covered lines and reports them as an XML document. In the MagicDraw plugin, this XML data is processed and displayed to the user as a coverage percentage and covered lines.

The tool is extended to support generation of contracts for python based web services, presented in Chapter 10. Different python testing tools can then be used to validate the web service implementation.

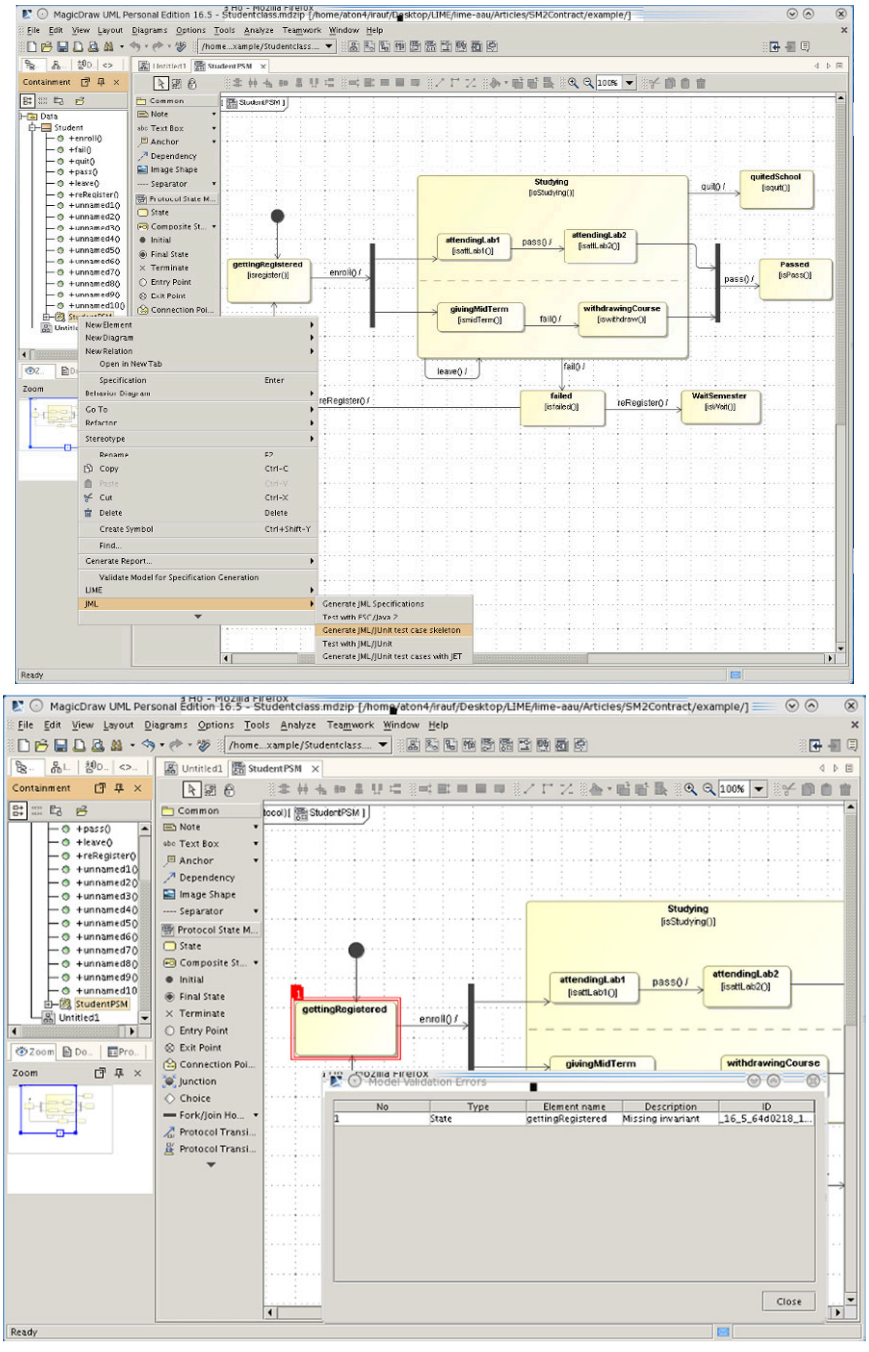


Figure 8.14: (Top) Screenshot of the MagicDraw plugin (Bottom) Validation example

8.6 Related Work

Our related work on validation is mainly divided in two areas: 1) Use of model checking techniques for validation of web service compositions 2) Use of contracts for testing.

8.6.1 Use of model checking techniques for validation

There is already a large body of work on using model checking techniques for validation and verification of web service compositions. Overviews of such works can be found in [114] and [31]. Mostly authors have used web service specific specification languages as their starting point and converted the specifications to an intermediate language that is accepted by model checking tools. Then, by taking advantage of the model checking tool capabilities they performed simulation, verification or test generation via model-checking. Most of these works use the selected model-checking tool only for simulation and verification; only a handful generate abstract tests from the verification conditions, and in most cases it is not clear how the abstract test cases (i.e., the counterexample traces) are transformed into executable ones and executed. In the following, we will revisit those works which are most similar to ours.

We can distinguish roughly two approach categories: those that target the PROMELA language [68] which is the input language for the SPIN model-checker [67], and those that target the UPPAAL timed automata which is the input language for the UPPAAL model-checker [24].

In the first category, the vast majority of approaches have used BPEL or OWL-S[89] for the specification of the web service composition. For instance, Garcia [59] generates test cases using test case specifications created from counterexamples that are obtained from model checking. The transition coverage criterion is used to identify transitions in BPEL specification that define the test requirements for producing test cases. These transitions are mapped to the model and expressed in terms of LTL property expressions. Test cases are generated using the test case specifications created from counter examples obtained from model-checking. Transition coverage is obtained by repeatedly executing the tool with each previously identified transition.

Fu. et al. [57] provide both bottom-up and top-down approaches to analyze the interaction between web services. In top-down approach, the desired conversation of a web service is specified as guarded automaton where guards of the automaton are XPath queries with LTL properties. The guarded automata are converted to PROMELA and used as input to SPIN model checker. The bottom-approach translates BPEL to guarded automaton and is used in similar fashion with SPIN model checker after translating guarded automaton to PROMELA. The web service conversation are analyzed for synchronization to verify their compatibility.

One distinct approach is given by Huang et al. [71]. They automatically

translate OWL-S specification of composite web service into a C-like specification language and Planning Domain Definition Language (PDDL) through a proposed integrated process. These can be processed with the BLAST model checker which can generate positive and negative test cases during model checking of a particular formula and test the web service using the test cases. They propose an extension to the OWL-S specifications and the PDDL to support their approach and use a modified version of the BLAST tool. Abstract, both positive and negative test cases are generated by formulating verification conditions for manually specified properties.

These works focus on BPEL4WS processes and OWL-S, this makes them dependent on specific execution languages for SOAP based services whereas our work is not dependent on implementation and supports REST architectural style. In addition, their work does not support requirement traceability and is not clear how tests are generated and executed. Furthermore, the works that use the PROMELA language for specification do not address real-time properties, due to the limited support for time in PROMELA.

In the second category, researchers have targeted timed automata specifications. In [39], Cambronero et al. verify and validate web services choreography by translating a subset of WS-CDL (Web Services Choreography Description Language) into a network of timed automata and then use UPPAAL tool for validation and verification of the described system. They also capture requirements by extending KAOS goal model and implement them. The work is supported by WST tool that provides model transformation of timed composite web services [38]. It takes UML sequence diagram and translates it to WS-CDL and then to UPPAAL for simulation and verification. In [50], Diaz et al. also provide a translation from BPEL4WS to UPPAAL timed automata. Time properties are specified in BPEL4WS and translated to UPPAAL. However, requirements are not traced explicitly, while verification and testing are not discussed.

Ibrahim and Al-Ani [72] transform BPEL specification to UPPAAL. The specification includes safety and security non-functional properties which are later formulated into guards in the UPPAAL model which could be similar to our verification of requirements. They do not consider neither real-time properties nor test generation.

Nawal and Godart [61] check the compatibility of web service choreography using model checking based approach that supports asynchronous timed communications. They use UPPAAL and provide full compatibility, partial compatibility and full incompatibility of web services. They propose a set of required abstractions for timed asynchronous communicating services that allow the use of model checker UPPAAL. Our work is somewhat similar to their work as we support time critical stateful REST webs service compositions using UPPAAL, however, in addition to verification we use UPPAAL with TRON to validate the implementation of web services.

Zhang [131] suggest the use of the temporal logic XYZ/ADL language [134]

for specifying web server compositions. They transform the specifications into a timed asynchronous communication model (TACM) and verify it using UPPAAL model checker.

In [79], uses BPEL specifications as a reference specification and transform them to an Intermediate Format(IF) based on timed automata and then propose an algorithm to generate test cases. Similar to our approach, tests are generated via simulation in a custom tool, where the exploration is guided by test purposes. One noticeable difference is that time properties are added manually to the IF specification, while we specify them at UML level.

These works provide approaches to verify and validate the service specifications by checking the properties of interest using UPPAAL tool, however our work, in addition to model checking the properties also performs conformance testing of the service composition via online model-based testing with the TRON tool and provides requirement traceability for non-deterministic systems.

8.6.2 Use of Contracts for Testing

A lot of work has been done on using contracts for testing. In [91], Meyer establishes the use of contracts to build reliable software components. Briand et al. [35] use and instrument contracts in code to ease the process of testing. Araujo et al. [20] further explore the use of contract for concurrent programs in the context of Java programs by extending JML specification language.

Ciupa and Leitner [45] have made use of Design by Contract assertions in their proposed solution of automatic testing and implemented as the Cdd Tool. The work of Leitner et al. in [84] expands on this work by merging the benefits of manual and automated testing in one technique i.e., AutoTest Experience. In [85], Leitner et al. introduces Contract Driven Development (CDD) as a new development method for testing. Contracts present in the code are used as test oracles and test cases are extracted from failure traces of program and failure runs.

The role of contracts for validating a composite web service is also less explored. In terms of using contracts for web service composition, we found the work of Milanovic [92]. In [92], Milanovic present a contract-based web service composition approach. In this work, he presents contract-based description language that is XML based and includes non-functional properties such as security, dependability, timeliness. The composition correctness is verified by modeling services as abstract machines.

[32] presents an approach for multi-party service composition based on contract using process calculi. They introduce the notion of subcontract relation that allow service composition in a manner that is deadlock and livelock free. Also, they relate their theory of contracts with theory of testing that can formally verify the composed service and also permits replacement of one service with another one without affecting the correctness of overall system.

In [47], provide automatic test case generation using contracts in web service

descriptions. They extend contracts with process control and other information and express them as OWL-S process specifications. These are then translated to Petri-Net models and test processes are generated using Petri-net behavior analysis. A decentralized framework for contract-based collaborative verification and validation of web services is presented in [21]. They propose a test-broker architecture in which all stakeholders of web service can contribute in improving the testing of the service. They explore the concept of design by contract and identify two categories of testing contracts including testing service contracts and test collaboration contracts.

All the works mentioned above take advantage of contracts present in the contract to perform different analysis and testing activities. In terms of exploring the idea of contracts at model level, work of [86] is noteworthy. They present a visual contract workbench tool that uses visual contracts for graphically specifying pre and post conditions of an operation. From these visual contracts JML assertions are generated for java classes to facilitate automatic monitoring of correctness of programs. Their model consists of class diagram and both pre and post conditions of visual contract are typed over it. The behavior of operations is given in terms of data state changes. Our work also addresses the concept of pre and post conditions of methods at model level. However, compared to their work, our approach does not only considers data state changes but also provide information on the sequence of method invocation and other dynamic behavior involving generation of pre and post conditions in different scenarios. Also, in case of REST web services, our resource model represented by a class diagram does not have method information. The information on allowed methods is generated from behavioral model that also provides information about pre and post conditions of methods, sequence of method invocations and other valuable information to create behavioral interfaces of REST web services.

8.7 Conclusion

In this chapter, we present our approach to verify the service design models and validate the service implementation. In our approach, a service can invoke other services and exhibit complex and timed behavior, while still complying with the REST architectural style. We have used UPPAAL model checker to verify the dynamic properties of our models. The service design models of the composite web service and its partner services are translated into UPPAAL timed automata which are verified for different dynamic properties with UPPAAL. To validate the service implementation, we generate tests using an online model-based testing tool, UPPAAL-TRON. The use of online model based testing proved beneficial as our system under test exhibits non-deterministic behavior due to concurrency and real-time domain.

Requirement traceability is also provided by tracing service requirements from

behavioral model to timed automata and their reachability is verified in UPPAAL. They are also used as test goals during test generation. Linking requirements to generated tests allowed us to quickly see which requirements have been validated and which not. In addition, our approach also provides edge and edge-pair coverage. The work is exemplified with a relatively complex worked example of a holiday booking web service and we provided preliminary evaluation results. The approach is validated using different benchmarking tools for UPPAAL and its efficiency is evaluated using code coverage tool and mutation testing.

In order to take advantage of contract-based testing approaches using contracts asserted in the code, we built a tool that automatically updates the Java file by inserting a contract derived from the UML protocol state machine. We used different testing tools like JML-JUnit and ESC/Java2 to show how contracts asserted in the code can be used to validate the behavior of the service.

Chapter 9

Implementation

The service design models of REST web services serve as specifications for a service developer. A service developer can study the design models and implement the service using them as reference document. However, lot of time and efforts can be saved by automatically generating code from the models as advocated by Model Driven Development (MDD) [99]. MDD advocates full code generation in bidirectional manner such that a change in one artifact reflects in the other making both the model and the code always consistent. We, however, provide a partial code generation tool in order to not clutter the design models with too much implementation details and at the same time facilitate the service developer to construct dependable web services using models. We generate code skeletons from the models that contain interface method contracts. The implementation of the interface methods can be done by the service developer.

In this chapter we demonstrate how web services are implemented as RESTful web services using our service design models. We also show the implementation of a service monitor that checks the interaction between a service and its clients and report if any of the parties breaches the interface contract. First we give a brief overview of the technologies we are using in our work. The implementation approach is discussed in section 9.2 and the implementation of service monitor is discussed in section 9.3. The implementation of service models is evaluated in section 9.4 and the chapter is concluded in section 9.5

9.1 Used Technologies

In this section, we present the languages and technologies that we used for the service implementation. Our service design models are in UML and our compiler is implemented in Python programming language [7]. We have chose python based Django web framework [66] to code and run our modeled web services.

9.1.1 Python

Python is a general-purpose, high-level programming language [7] created by Guido van Rossum in the early 90's. It is a language similar to Perl, but with a very clean syntax that offers a readable code.

Here we present some the most important characteristics:

- Scripting language: An interpreted language or scripting is one that is run using an intermediate program called interpreter rather than compiled program in machine language code that you can understand and run in a computer directly (compiled languages). The advantage of compiled languages is that their execution is faster. However, interpreted languages are more flexible and more portable.
- Duck typing: We say that a language supports duck typing if an object of particular type is compatible with a function when it provides all the methods or method signatures that are requested from it by the method at runtime [3]. Duck typing is heavily supported in Python.
- Strongly typed: It is not allowed to treat a variable as being of a different type from what it has. It is necessary to convert the variable to a new type before using it as such. For example, if we have a variable containing a string, it cannot be treated as a number ("9"). In other language the type of the variable change to accommodate the expected behavior, although this is more prone to errors.
- Cross-platform : The Python interpreter is available on many platforms (UNIX, Linux, DOS, Windows, Solaris, OS/2, MacOS). So if you do not use specific platform libraries, you can run your program in all these systems without major changes.
- Object-oriented (OOP): This is a programming paradigm in which real-world concepts of a problem could be represented in classes and objects in our program. The execution of the program is a series of interactions between objects.

9.1.2 Django Web Framework

Django [66] is an open source web application framework, written in Python, which follows the model-view-controller architectural pattern (MVC). Django is developed with the intention of easing the creation of complex, database-driven websites. It emphasizes reusability and "pluggability" of components, rapid development, and the principle of DRY (Don't Repeat Yourself). Python is used throughout, even for settings, files, and data models. Django also provides an optional administrative CRUD (create, read, update and delete) interface that is generated dynamically through introspection and configured via admin models.

Here are some of the important features of Django [66]:

- Object-relational mapper: Data models can be defined entirely in python.

You can either use dynamic database-access API that comes with it or write SQL if needed.

- Automatic admin interface: Django provides a production ready admin interface that can let you add and update content.
- Elegant URL design: It lets you design crud-free URLs with no framework-specific limitations.
- Template system: With Django template system, you can separate design, content and python code.
- Cache system: The cache system helps you save the result of some expensive computation for future reference so that you don't have to perform it again. The feature optimizes the performance. Django provides a robust cache system offering different levels of cache granularity.
- Internationalization: Django has full support for multi-language applications, letting you specify translation strings, and providing hooks for language-specific functionality. Django has full support for translation texts and allows developers to specify which parts of their application would be translated or formatted for local languages using translation strings.

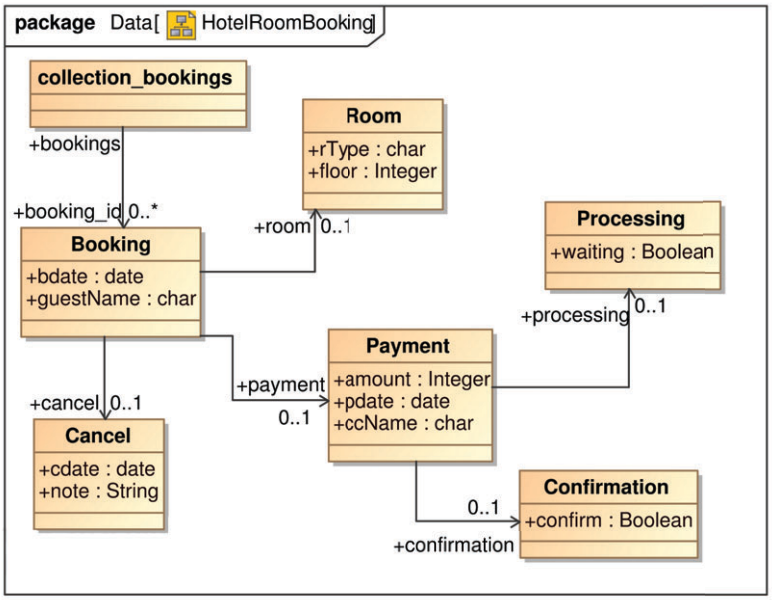
9.2 Implementation

The implementation of service design models consist of three important parts, i.e., 1) Developing the service design models that are input to the compiler, 2) Writing the python compiler that processes the information in the models and 3) the Django file results that are the output of the compiler.

9.2.1 Service Design Models

We have represented the static and dynamic structure of REST web service using UML class and state machine diagram. We have explained in detail the construction of our resource and behavioral models in Chapter 3 and Chapter 4, respectively, with the help of hotel room booking service example. The user of the service books a room and pays for it. While a third party service processes the payment, the service waits for the processing and marks the booking as paid once the confirmation is received. The booking can be canceled anytime if it is not waiting for the payment processing.

We reproduce the same example in this chapter to demonstrate how the service is implemented in Django web framework. Figure 9.1 and Figure 9.2 show the resource and behavioral models for hotel room booking RESTful service. The standard HTTP methods are called on the service to navigate through the different states of hotel booking service. Every piece of information that user can use, e.g., cancelation, payment and booking etc. is accessible via independent URIs. Also, information about when a method should or should not be invoked, e.g., making a booking cancel request, can also be inferred from the models.




```
/bookings/{bid}/
/bookings/{bid}/cancel/
/bookings/{bid}/payment/
/bookings/{bid}/room/
/bookings/{bid}/payment/processing/
/bookings/{bid}/payment/confirmation/
```

Figure 9.1: (Top) Resource Model for HRB RESTful Web Service. (Bottom) Resource paths

The well-formedness rules for the models have been explained earlier. We have imposed the following restrictions as well to facilitate the implementation.

> The *root resource definitions* must always be collection resources and their name starts with *collection_*. In our example we have one collection *resource definition*, *collection_bookings*. We can access all system *resource definitions* through this collection *resource definition*.
> The primary key of *resource definitions*, if modeled, must be written as follows: resource_name + "_id ". This decision is taken by an agreement to facilitate the implementation and understanding.
> The association that goes from the *root resource definition* to its contained *resource definition* must be marked with the name of the primary key of that *resource definition*, i.e., resource_name + "_id ".

From the diagram we can see the paths in which each *resource definition* can

be accessed. At the bottom of Figure 9.1 we can see the paths for this example. We can see that every path starts from *root resource definition*.

We have used MagicDraw UML as a modeling tool to model our example. We generate XML Metadata Interchange (XMI) of the behavioral model from this tool which is saved into a file.

9.2.2 Python Compiler

This section details how a semi-automatic translator is created. The tool takes as input XMI of the models. All the required information is retrieved from the XMI of models and processed. Compiler collects relevant information from resource and behavioral models, treats it and creates internal data structures, in order to supplement the information between diagrams. We can divide the process into three phases:

1. Gather the necessary information from the input models.
2. Analyze information, create appropriate data structures and supplement this information with both the diagrams.
3. Export all the information to code, creating the file structure needed to run the system for Django Framework.

In the final result, we obtain the necessary Django project files. The three main required files are models.py, urls.py and views.py. Below we explain what steps we follow to obtain them.

Phase 1:

In the first phase we take as input the XML [4] files of the diagrams. We are interested in gathering information for three main files of Django application, i.e, model.py, urls.py and views.py files. These files provide different information:

- models.py: Contains the information about the system database.
- urls.py: Contains the URL information that can be called on the service and their mapping to associated views.
- views.py: Provides functional details. Each view is responsible for doing one of the two things: returning an HttpResponse object containing the content for the requested page, or raising an exception such as Http 404.

First we will obtain the information necessary to create the models.py file. This file contains the information of the system database. To do this we look for the resources in the resource model. For each resource in the resource model, we create a table in the database and analyze its associations to complete its relationships with the right foreign keys. We do not merge tables since we consider it a job without reward.

We then move forward to collect information for the next file, urls.py. This maps the relative URLs of each resource to their respective views. We find the information of URL paths from the resource model. We use the rolenames of

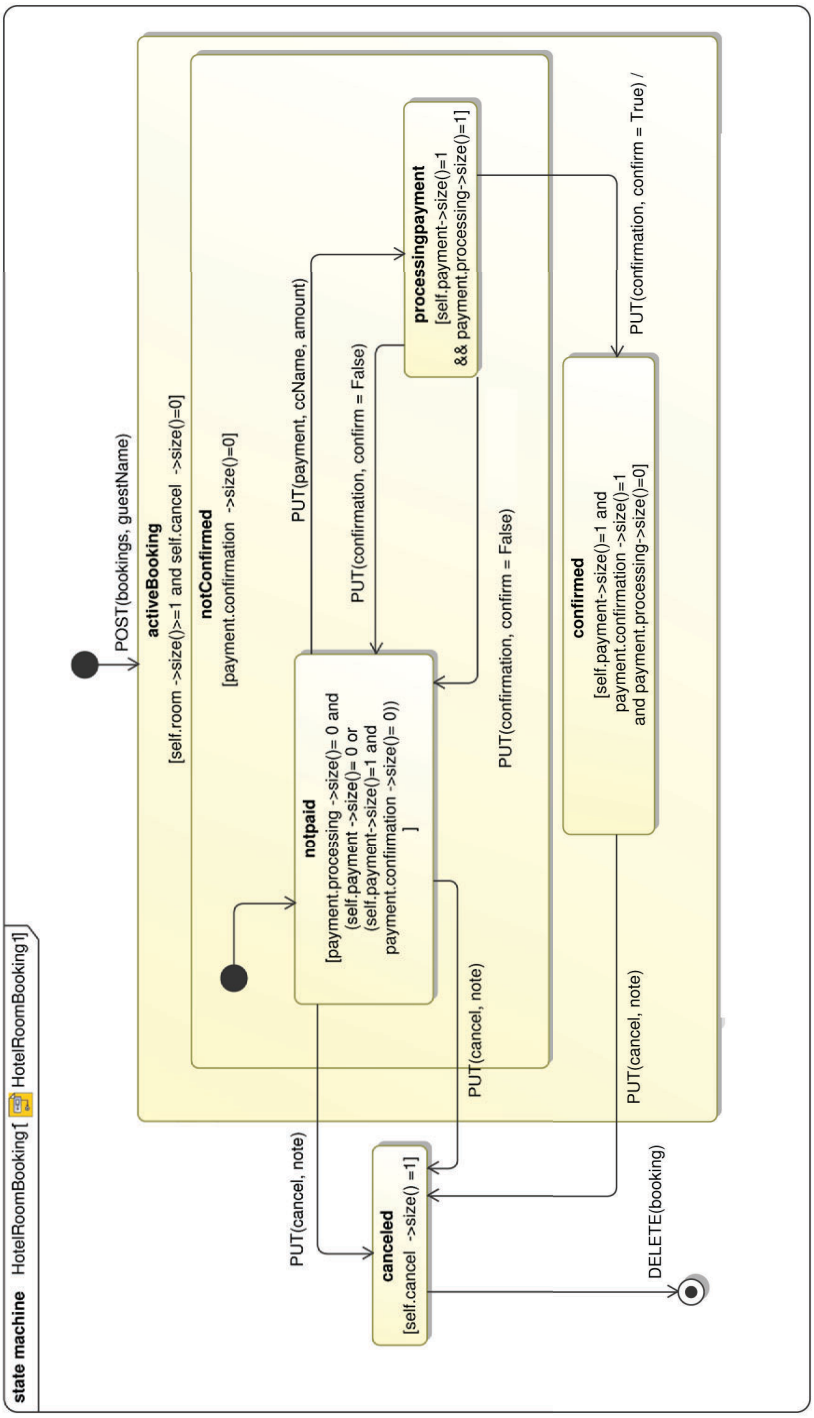


Figure 9.2: Behavioral Model for HRB RESTful Web Service

associations to compose the paths of each resource, always starting from the *root*, and particularly if we are referencing an item in the collection.

Finally we started looking for information for the views.py file which one of the most important file for Django. This file contains all the functionality of the system and the code we will run when accessing a resource through its URL based the allowed requests (GET, PUT, POST, and DELETE). This information is inferred from the behavioral model. It provides the necessary information about each resource, what methods can be executed on it and when to trigger it, i.e, whether certain preconditions and postconditions are true. For the extraction of method contracts from behavioral model, we relied on our work on generating contract from protocol state machine that we have presented in section 8.1.

Phase 2: Once we have our tree data structure, we must complete it and create the necessary files correctly. Specifically we need to complete the information in urls.py with the information in views.py. Moreover we should take care of some special situations, i.e., how to handle and implement the logic of the views for each resource and transition. These special situations include cases like transitions with the same method from two different source states to the same target state, two or more transitions from the same source state to different target states with distinct preconditions and postconditions and two or more transitions from different source states to different target states.

Phase 3:

In the final phase, we have all the information properly code structured to create the desired language or platform. In this project we are exporting a Django project and creating the corresponding files.

The first files we create are those needed to run the project. We simply put the name of the project, which URLs it is going to use, and indicate where our application is. After this we complete the models.py file that contains the tables of the database. In this file we place a table for each resource, indicating each time the corresponding primary and foreign keys, and the attributes that are specified in the model.

The models.py file for our hotel room booking service is:

Listing 9.1: Implementation of Database Models for HRB Service

```
from django.db import models

class Booking(models.Model):
    bDate =  models.DateTimeField()
    guestName = models.CharField(max_length=200)

class cancel(models.Model):
    booking = models.ForeignKey(booking)
    note = models.CharField(max_length=200)
    cdate = models.DateTimeField()

class Payment(models.Model):
    booking = models.ForeignKey(booking)
```

```
    amount = models.IntegerField()
    pDate = models.DateTimeField()
    ccName = models.CharField(max_length=200)


class Processing(models.Model):
    booking = models.ForeignKey(booking)
    waiting = models.BooleanField(default=False)


class Confirmation(models.Model):
    booking = models.ForeignKey(booking)
    confirm = models.BooleanField(default=False)

class Room(models.Model):
    booking = models.ForeignKey(booking)
    rType = models.CharField(max_length=200)
    floor = models.IntegerField()
```

After that we create the urls.py file with all the relative URLs of the views in the project. In this way we can access the service resources. The urls.py file generated for hotel room booking REST service is give below:

Listing 9.2: Implementation of URLS for HRB Service

```
from django.conf.urls.defaults import *
from myApp.views import *

urlpatterns = patterns('',
  (r'^collection_bookings/$' ,  bookings_collection),
  (r'^collection_bookings/(\d{1,3})/$' ,  bookings_booking_detail),
  (r'^collection_bookings/(\d{1,3})/room/$' ,  bookings_rooms_detail),
  (r'^collection_bookings/(\d{1,3})/cancel/$' ,  bookings_cancellation
      ),
  (r'^collection_bookings/(\d{1,3})/payment/$' ,  bookings_payment),
  (r'^collection_bookings/(\d{1,3})/payment/processing/$' ,
      bookings_pwaiting),
  (r'^collection_bookings/(\d{1,3})/payment/confirmation/$' ,
      bookings_pconfirmation),
)
```

The last file we create is views.py. This file contains view for each allowed method on resource with the correct logic. These views consist of preconditions and postconditions, the main action of the method, and returns the proper HTTP code. As we do not know from the model what to do in some cases, we have written the action as an skeleton. Here is the user interacts to complete such functions.

As an example, lets look at the functionality implemented as views for *payment* resource. The first view $booking\_payment(request, booking\_id)$ in Listing 9.3 shows implementation of *payment* resource. The behavioral model in Figure 9.2 shows that the allowed methods for this resource are GET and PUT. These two methods are listed in the list of allowed methods in *booking_payment* view and each incoming request to this view is first verified to be one of these methods, otherwise an HTTP response of method not allowed is given. The request is redirected to the view that corresponds to the invoked method. If the invoked

method is GET, it goes to *booking_payment_get* view and if it is a PUT method then the request is redirected to *booking_payment_put*. These views contain the code that implements the logic and interacts with the database to perform the required task.

Listing 9.3: Payment View

```
def booking_payment(request, booking_id):
    if not request.method in ["GET", "PUT"]:
        return HttpResponseNotAllowed(["GET", "PUT"])
    if request.method == "GET":
        bid = bid
        return booking_payment_get(request, booking_id)
    if request.method == "PUT":
        bid = booking_id
        amnt = request.POST.get('amnt')
        ccName = request.POST.get('ccName')
        return booking_payment_post(request, bid, amnt, ccName)

def booking_payment_get(request, bid):
       p = payment.objects.filter(booking=bid)
       if p:
           json = serializers.serialize("json", p)
           return HttpResponse(json, mimetype="application/json")
       else:
           return None

def booking_payment_put(request, bid, amnt, ccName):
       p = bookings_payment_get_local(booking_id)
       conf = bookings_confirmation_get_local(booking_id)
       proc = bookings_processing_get_local(booking_id)
       b = bookings_booking_detail_get_local(booking_id)
       c = bookings_cancel_get_local(booking_id)
       r = bookings_room_get_local(booking_id)
       if not p:
          pre_p = False
       else:
          pre_p = True
       deserialized = serializers.deserialize("json", b)
       b_detail = list(deserialized)[0].object
       a = []
       for field in ["bDate", "cancel", "cancel_note", "room", "gName"
           ]:
           new_val = getattr(b_detail, field, None )
           a.append(new_val)
       if  b and r and not p and not proc and not conf and not c and a
           [4]==ccName:
           now = datetime.datetime.now()
           cc = ccName
           a = amnt
           p = payment(confirm=False, pDate=now, waiting=False, amount
               =a, p_try=0, ccName = cc, booking_id=bid)
           p.save()
       b = booking_detail_get_local(bid)
       r = room_detail_get_local(bid)
       c = booking_cancel_get_local(bid)
       pc = booking_pconfirmation_get_local(bid)
       post_p = booking_payment_get_local(bid)
       if  b and r and not pre_p and post_p and not conf and not proc
           and not c:
```

```
        response = HttpResponse("created")
        response.status_code = 201
        return response
    else:
        response = HttpResponse("not created")
        response.status_code = 406
        return response
```

At the end we obtain a complete Django Web Framework project that implements our RESTful Web Service.

Some of the main features of the compiler are:

- It is written in Python 2.7
- We use XML 2.1 and UML 2.0
- Requirement of lxml [11] module

Using the compiler is very simple:

```
uml2django ProjectName DiagramsFileinXML
```

where ProjectName denotes the name of our project in Django and DiagramsFileinXML will contain the diagrams required in XML format.

9.2.3 Django Files Result

The result of the compiler is a project in Django web framework with all files necessary for execution. In order to do so, we first we find the files necessary for running the program, i.e.:

- __init__.py: Tells Python that the directory is a python module and can be imported (and imported from).
- settings.py: Django settings file contains all the configuration of the Django installation.
- manage.py: It is the first file to be executed. It calls settings.py file to start.
- urls.py: This specifies all the URLs of the different applications in the project.

A Django project could contain many several applications. Each one of them represent a different service. Each application has main three files: models.py, urls.py, views.py.

Once seen how the Django project is, we continue with fine tuning the project. As mentioned above, we need the intervention of the user in some parts of the views.py file. The user must modify the skeleton methods with the desired code.

At the end of completing the code, we run our project. To do this we boot the server and create the database to store our resources properly. When finished, we are ready to test the web service implementation and make sure it works properly.

Users can use cURL to invoke URIs if they want to use the service. cURL is a command line tool that is a capable HTTP client and supports most of HTTP methods, authentication mechanisms, headers etc. [2]. For invoking a PUT method

on *payment* resource with *amnt* value, on local server, the following command can be used on cURL:

$curl - X\ PUT - d\ amount = 115\ - d\ ccName ='' Thomas''\ http://127.0.0.1:8000/\ bookings/3/payment/$

Alternatively, users can also use REST Client available as a plugin for different browsers like Mozilla Firefox and Chrome.

9.3 Implementation of a Service Monitor

A service monitor can be used to continuously verify the functionality of an implemented web service. This monitoring mechanism can keep a check on the behavior of both the client and the provider. The client is checked for invocation to the service under right conditions and the provider of the service is constraint to provide the implementation as specified.

The monitoring mechanism can be implemented in Django by using the behavioral information present in our behavioral model. The service monitor is implemented as a service proxy. It listens for requests from the client, verifies the conditions to invoke the method and then forward it to the actual service implementation.

The behavioral model provides a behavioral interface that can be published with the service as a specification. This gives information about the conditions in which a method should be invoked on its interface and also about its expected conditions. The specification of a service interface can be used to build a proxy interface to test the functionality of that service and to invoke the service in right conditions.

In this section we show how we have implemented a proxy interface for holiday room booking service detailed above. In proxy interface, a method is implemented for each of the methods that are invoked on the REST web service interface using urllib2. urllib2 is a python module that is used to fetch URLs [8]. In a proxy interface for holiday room booking service, a GET method on payment resource, for example, is implemented as:

Listing 9.4: Excerpt of GET view in Proxy Interface

```
def booking_payment_get(request, bid):
    req = urllib2.Request('http://127.0.0.1:8000/bookings/\%s/payment/
        ' \% bid)
    try:
        response = urllib2.urlopen(req)
        the_page = response.read()
        return HttpResponse(the_page)
    except:
        return HttpResponse(status=404)
```

Each GET view returns an HTTP response object. When a POST, PUT or DELETE method is implemented in the proxy interface, it manipulates the status codes of the HTTP response objects and asserts them as method pre and post conditions. An excerpt of holiday room booking service proxy interface that shows a PUT method on the payment resource is given as follows:

Listing 9.5: PUT method on Payment in the Proxy Interface

```
def booking_payment_put(request, bid, amnt, ccName):
    b = booking_detail_get(request, bid)
    r = room_detail_get(request, bid)
    c = booking_cancel_get(request,bid)
    p = booking_payment_get(request, bid)
    pc = booking_confirmation_get(request, bid)
    pr = booking_processing_get(request, bid)
    if not p.status_code == 200:
        pre_p = False
    else:
        pre_p = True
    if  b.status_code = 200 and r.status_code == 200 and p.status_code
     == 404 and pc.status_code == 404 and pr.status_code == 404
        and c.status_code == 404:
        values ={'amnt': 33, 'ccName': 'Thomas'}
        mydata = urllib.urlencode(values)
        opener = urllib2.build_opener(urllib2.HTTPHandler)
        request = urllib2.Request('http://127.0.0.1:8000/bookings/%s/
            payment/' % bid, data=mydata)
        response = urllib2.urlopen(req)
        the_page = response.read()
    else:
        return = HttpResponse(status=404)
    post_p = booking_payment_get(request, bid)
    if  b.status_code = 200 and r.status_code == 200 and pc.
        status_code == 404 and pr.status_code == 404 and c.status_code
     == 404 and not pre_p and post_p.status_code == 200:
        return HttpResponse(the_page,status=201)
    else:
        return HttpResponse("not created",status=406)
```

9.4 Evaluation

In this section we reflect on the decisions taken, see the results and analyze positive and negative aspects of it. To do this we must define the parameters of how to study the solution of our compiler. For example:

- Did we get a translation which fully reflects the model?
- How good are the restrictions we have assumed in the models?
- Is creating a python compiler the best solution for this process? Why not others?

In Model Driven Development (MDD) [99], we try to achieve reliable and accurate results for a given platform or language from the models. This process may not always be a fully automated and there may not be a full equivalence

between the model and the code obtained, as in our case. This is because the lack of properties or expressiveness in the model entails the results with lack of information. We require user intervention to fill in the missing lines of code in code skeleton that we have generated automatically in order to avoid cluttering of too much information in the models that may make the models complex.

Some of the restrictions that we have introduced on the models are due to the fact that the developers need some mechanism to detect relationships or dependencies between resources and other elements. In our design restrictions, the user has to use certain names to detect certain information. This can be bad as it can lead to error if we do not take into account the modeling guidelines. It is a compromise solution, which should be studied for future improvements.

For the code generation from the models, we can find different ways to tackle the challenge. Transformation languages like ATL or QVT can facilitate this process. These languages are very useful if we desire an equivalence between the model and the code, and also if the result is automatic requiring no extra processing or analysis. Instead in our implementation, we chose to create a python compiler with greater capacity for compilation and processing of data structures so that we can analyze different parts of the code.

9.4.1 Advantages

The purpose of these tools is to make life easier for the developer. You can generate code for a particular language or platform through simply modeling a problem or system without writing any code. This is the main advantage in developing such tools.

Another very important point to develop such tools would be to obtain code without inconsistencies between the model and the code since it is an automated process which does not involve manual interventions.

Time is another important factor for a programmer. To implement applications in an agile way, changing just one part of the code allow the developer to devote more time to testing and different trials. This is another good reason and motivation for the creation of such tools.

9.4.2 Disadvantages

Not all of this is the panacea of software development. To begin implementing the compiler, there are always new problems and issues to discuss. Some of them force you to make wrong decisions, and not always choose the best solution. It may be the lack of time to develop the application or because the timing of the decision not occurred to us in time. One decision that could enter into a disadvantage is that we implemented the interface as skeleton code, and the user has to complete it afterwards because our primary focus was the implementation of REST interface. One of the major difficulties that we faced was to determine which views are

needed for Django project in addition to the diagrams. We must make complete pairs of intermediate process information through the views and the URLs.

9.5 Conclusion

In this chapter, we have demonstrated how our service design models are implemented in Django web framework. We have provided a semi automatic code generation approach. The developers of the systems can fill in the missing code as required. The tool is implemented using Python and Django web framework. The resource and URL information is extracted from the resource models and implemented in models.py and urls.py files, respectively. The information on methods and their contracts is extracted from the behavioral model and asserted in views.py. The URL information from urls.py and function information from views.py are mapped together to redirect URLs to appropriate functionality. We also show the implementation of a service monitor via a proxy interface that can continuously verify the functionality of an already implemented service. We also evaluated the implementation approach for its merits and demerits.

Chapter 10

Conclusion

In this thesis, we have presented a model-driven approach to design and validate web services with stateful and timed behavior that exhibit REST interface features. The goal of the thesis is to facilitate the service developer in the creation of REST web services for advance scenarios by providing a holistic approach that spans through different phases of service development. The conclusion we discuss here is categorized according to the different research areas that we have worked in and answers the research questions that we have posed in these areas.

10.1 Design

We have given an approach to design REST web services and their compositions for advanced scenarios offering stateful behavior. The created web services are REST compliant such that they exhibit REST interface features of *addressability*, *connectedness*, *statelessness* and *uniform interface*. The interfaces of REST web services and their compositions are modeled using UML class and state machine diagrams. The composition process is modeled with activity diagram and scenario models. The design models also provide information about the domain specific requirements, time restrictions and authorized users that facilitate the service developer to implement the right functionality.

The service design models provide behavioral REST interfaces that also provide information on how to use the service correctly. The approach to generate behavioral interfaces is first applied to a class with the help of UML class and protocol state machine diagrams. The behavioral information is inferred from them for all possible cases of UML protocol state machines. A prototype tool is developed to generate code skeletons with method contracts from UML protocol state machines. The approach is then applied to the service design models to create and implement behavioral interfaces for a RESTful web service.

The design approach is first demonstrated with a pedagogical example of hotel room booking service to explain the concepts of our design approach. It is then

applied on a relatively complex worked example of holiday booking composite RESTful web service from industrial context that shows how timed web services with stateful behavior in complex scenarios can be built using our design approach. To the best of our knowledge, our model-driven approach to design behavioral REST web service interfaces for advanced scenarios has not been addressed before.

10.2 Consistency Analysis

The problem of checking the consistency of models is addressed using semantic web technologies. By using semantic web technologies for the consistency checking of our design models we not only take advantage of several efforts done previously to reason ontologies, i.e., to derive facts from them, but also provide a mechanism for the semantic representation of REST interfaces that can be part of the semantic web. Our work provides a way to represent the service design models of a REST web service as an OWL 2 ontology and use ontology reasoners to check it for any unsatisfiable concepts resulting in service implementations with faulty behavior. The approach is fully automated thanks to the implemented translation tool and the existing OWL 2 reasoners. We have also evaluated the performance of this approach using both valid and mutated models consisting of 10 to 2000 model elements. They are evaluated on the basis of UML to OWL 2 translation time, and the reasoning time taken by OWL 2 reasoners. The result showed that the translation and reasoning time on all the models was less than 4.5 seconds in all cases. This shows that the approach can process relatively large UML models in few seconds.

10.3 Validation

The validation of service design models and their implementation is done using UPPAAL model checker. We have implemented a translation tool that translates design models to UPTA. The service design models are verified for their basic properties of the models like deadlock freedom, liveness, reachability and safety using UPPAAL model checker. The service implementation is validated with a model-based black-box conformance testing tool, i.e., UPPAAL-TRON. The approach also provides requirement traceability, which is an important part of our work. By using requirement traceability, whenever a test fails, we can trace-back the parts of the models from which the failure originated based on the requirements covered by that test. We have applied our validation approach on a relatively complex holiday booking composite RESTful web service.

For benchmarking the verification process, we have used the `verifyta` command line utility of UPPAAL for verification of the specified 5 properties. We have used the `memtime` tool to measure the time and memory needed for verification. The result showed on average 0.20 seconds and 54996 KB of memory

being used. This shows that the current size models leave room for scalability of the approach. In order to evaluate the efficiency of our approach, we compared the specification coverage with the code coverage yielded by a given test run and the results showed 100% edge coverage with 78% code coverage that was quite promising since we did not model the negative cases. To evaluate the fault detection capabilities of our approach, we manually created 30 mutated versions of the original program code of our worked example. The faults are seeded in those parts of the code that are covered with 100% edge coverage of the model. Assuming that the original version of the composite web service is the correct one, we ran 100 test sessions in TRON against it. For each mutated version of the composite web service, we set the TRON to execute 100 test sessions against it. Out of the 30 mutated programs, 28 mutants were killed and 2 were alive.

By benchmarking of various features of our testing approach and analyzing the verification and validation results of our worked example, we have demonstrated the applicability of our validation approach and its practicality in real world situations.

10.4 Implementation

We have implemented a partial code generation tool in Python that generates code skeletons from the service design models in Django web framework. The generated code contains pre-conditions and post- conditions for each method and the developer only needs to manually input the functionality of the methods. We have not developed a full code generation tool since we focused primarily on the interface concerns of the web service. All our worked examples are implemented using our implementation tool.

We advocate that the web service created using our approach is REST compliant and can be trusted for the functionality it advertises. The created web services offer the properties of REST architectural styles that make them scalable, extensible and allow them to play well with the existing tools and infrastructure of the web. The uniform interface requirement (use of standard HTTP methods) and the connectedness requirement (creation of connected resource graph and hyperlinks) in our approach allows the use of existing web tools and infrastructure like web crawlers, curl, proxies and caches. The addressability requirement (especially when using hierarchical addresses) would lead to extensible web services and the statelessness requirement allows the development of systems that can handle many service requests simultaneously.

Our approach to design and validate REST web services is novel. The approach is also fully automated thanks to the tools that we have implemented and those already available in the industry. The service developer can model the system graphically using our approach and be positive about the fact that the services created using them will exhibit REST interface features. Consequently, the developer is supported with different tools in various stages of the development cycle to

create dependable REST web services.

The next step of our work is to integrate different translation tools behind one interface. One limitation of our approach is to keep the models up to date with the running system during its life cycle and evolution. We plan to address this limitation in our future work and study how to quantify the efforts needed to keep the models updated as the system evolves. In the future, we also plan to address the complexity of models. For that we plan to split the models into several parts, via slicing or aspect oriented approaches, each focusing on a different concern of the system. Nonetheless, the approach provides a promising approach to develop REST web services for complex scenarios that can be trusted for their functionality.

Bibliography

[1] Code coverage measurement for Python – coverage, v. 3.6. https://pypi.python.org/pypi/coverage. Accessed: 20.08.2013.

[2] cURL. http://curl.haxx.se/. Accessed: 20.08.2013.

[3] Duck Typing. http://c2.com/cgi/wiki?DuckTyping. Accessed: 12.12.2013.

[4] Extensible Markup Language (XML). http://www.w3.org/XML/. Accessed: 01.12.2013.

[5] HTTP Authentication. http://www.httpwatch.com/httpgallery/authentication/. Accessed: 20.08.2013.

[6] Nomagic MagicDraw webpage at http://www.nomagic.com/products/magicdraw/. http://www.nomagic.com/products/magicdraw/. Accessed: 18.11.2012.

[7] Python programming language. http://python.org/. Accessed: 18.06.2013.

[8] urllib2 - extensible library for opening URLs. *Python Documentation*. Accessed: 18.10.2012.

[9] Web Services Directory. http://www.programmableweb.com/apis/directory. Accessed: 2014-05-02.

[10] Web services resources framework (wsrf 1.2). https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrf. Accessed: 2013-11-01.

[11] xml, XML and HTML with Python. http://lxml.de/. Accessed: 20.08.2013.

[12] Rosa Alarcon and Erik Wilde. Linking Data from RESTful Services. In *Third Workshop on Linked Data on the Web, Raleigh, North Carolina*, 2010.

[13] Rosa Alarcon, Erik Wilde, and Jesus Bellido. Hypermedia-driven RESTful service composition. In *Service-Oriented Computing*, pages 111–120. Springer, 2011.

[14] Subbu Allamaraju. *RESTful Web Services Cookbook*. O'Reilly, 2010.

[15] Rajeev Alur, Costas Courcoubetis, and David Dill. Model-checking for real-time systems. In *Proceeding of Fifth Annual IEEE Symposium on Logic in Computer Science, LICS,*, pages 414–425. IEEE, 1990.

[16] Rajeev Alur and David Dill. Automata for modeling real-time systems. In *Automata, languages and programming*, pages 322–335. Springer, 1990.

[17] Thomas Ambuhler. *UML 2.0 Profile for WS-BPEL with Mapping to WS-BPEL*. Universitat Stuttgart, 2005.

[18] Jim Amsden, Tracy Gardner, Catherine Griffin, and Sridhar Iyengar. Draft UML 1.4 Profile for Automated Business Processes with a mapping to BPEL 1.0. *Draft, IBM UK Laboratories, Hursley Park*, 1, 2003.

[19] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1. May 2003. http://www.ibm.com/developerworks/.

[20] Wladimir Araujo, Lionel Briand, and Yvan Labiche. Concurrent contracts for Java in JML. In *19th International Symposium on Software Reliability Engineering (ISSRE)*, pages 37–46. IEEE, 2008.

[21] Xiaoying Bai, Yongbo Wang, Guilan Dai, Wei-Tek Tsai, and Yinong Chen. A framework for contract-based collaborative verification and validation of web services. In *Component-Based Software Engineering*, pages 258–273. Springer, 2007.

[22] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, Kannan Govindarajan, Alan Karp, Harumi Kuno, Mike Lemon, Gregory Pogossiants, Shamik Sharma, et al. Web services conversation language (wscl) 1.0. *W3C Note*, 14, 2002.

[23] IBM BEA. Microsoft: Web services transactions (ws-transactions), 2002.

[24] Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, John Hakansson, Paul Petterson, Wang Yi, and Martijn Hendriks. UPPAAL 4.0. In *Quantitative Evaluation of Systems, 2006. QEST 2006. Third International Conference on*, pages 125–126. IEEE, 2006.

[25] Jesus Bellido, Rosa Alarcón, and Cesare Pautasso. Control-Flow Patterns for Decentralized RESTful Service Composition. *ACM Transactions on the Web (TWEB)*, 8(1):5, 2013.

[26] Ed Benowitz, Ken Clark, and Garth Watney. Auto-coding UML statecharts for flight software. In *Second IEEE International Conference on Space Mission Challenges for Information Technology*, pages 5–pp. IEEE, 2006.

[27] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext transfer protocol–HTTP/1.0, 1996.

[28] Dag Björklund, Johan Lilius, and Ivan Porres. Towards Efficient Code Synthesis from Statecharts. In *Workshop of the pUML-Group held together with the «UML» 2001 on Practical UML-Based Rigorous Development Methods-Countering or Integrating the eXtremists*, pages 29–41. GI, 2001.

[29] Motick Boris, Patel-Schneider Peter F, and Cuenca Grau Bernardo. *OWL 2 Web Ontology Language Direct Semantics*.

[30] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, et al. Web services addressing (WS-Addressing), 2004.

[31] Mustafa Bozkurt and other. Testing web services: A survey. *Department of Computer Science, King's College London, Tech. Rep. TR-10-01*, 2010.

[32] Mario Bravetti and Gianluigi Zavattaro. Contract based multi-party service composition. In *International Symposium on Fundamentals of Software Engineering*, pages 207–222. Springer, 2007.

[33] Mario Bravetti and Gianluigi Zavattaro. A theory for strong service compliance. In *Coordination Models and Languages*, pages 96–112. Springer, 2007.

[34] Mario Bravetti and Gianluigi Zavattaro. A foundational theory of contracts for multi-party service composition. *Fundamenta Informaticae*, 89(4):451–478, 2008.

[35] Lionel C Briand, Yvan Labiche, and Hong Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. In *ACM SIGSOFT Software Engineering Notes*, volume 27, pages 70–80. ACM, 2002.

[36] Manfred Broy, María Victoria Cengarle, Hans Grönniger, and Bernhard Rumpe. Considerations and Rationale for a UML System Model. *UML 2 Semantics and Applications*, pages 43–60.

[37] Felipe Cabrera, George Copeland, Tom Freund, Johannes Klein, David Langworthy, David Orchard, John Shewchuk, and Tony Storey. Web services coordination (ws-coordination). *joint specification by BEA, IBM, and Microsoft*, 2002.

[38] M Emilia Cambronero, Gregorio Díaz, Enrique Martínez, Valentín Valero, and Llanos Tobarra. WST: a tool supporting timed composite Web Services Model transformation. *Simulation*, 88(3):349–364, 2012.

[39] M Emilia Cambronero, Gregorio Díaz, Valentín Valero, and Enrique Martínez. Validation and verification of Web services choreographies by using timed automata. *Journal of Logic and Algebraic Programming*, 80(1):25–49, 2011.

[40] Xiaoxia Cao, Huaikou Miao, and Qingguo Xu. Verifying Service-Oriented Requirements Using Model Checking. In *Proceedings of the 2008 IEEE International Conference on e-Business Engineering*, ICEBE '08, pages 643–648, Washington, DC, USA, 2008. IEEE Computer Society.

[41] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *ACM SIGPLAN Notices*, volume 43, pages 261–272. ACM, 2008.

[42] Yoonsik Cheon. Automated random testing to detect specification-code inconsistencies. *Departmental Technical Reports(CS)Paper 101*, 2007.

[43] Yoonsik Cheon and Gary T Leavens. The JML and JUnit way of unit testing and its implementation. *Rap. tech*, 2004.

[44] R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. June 2007. www.w3.org/TR/wsdl20/.

[45] Ilinca Ciupa and Andreas Leitner. Automatic testing based on design by contract. In *Proceedings of Net. ObjectDays*, volume 2005, pages 545–557. Citeseer, 2005.

[46] Edmund M Clarke, Orna Grumberg, and David E Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1512–1542, 1994.

[47] Guilan Dai, Xiaoying Bai, Yongbo Wang, and Fengjun Dai. Contract-based testing for web services. In *31st Annual International Computer Software and Applications Conference, COMPSAC*, volume 1, pages 517–526. IEEE, 2007.

[48] Doug Davis, Ashok Malhotra, Oracle Katy Warr, and Wu Chou. Web Services Transfer (WS-Transfer). *World Wide Web Consortium, Recommendation REC-ws-transfer-20111213*, 2011.

[49] Birgit Demuth and Claas Wilke. Model and Object Verification by Using Dresden OCL. In *Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice,*, pages 81–89, 2009.

[50] Gregorio Dıaz et al. Model Checking Techniques applied to the design of Web Services. *CLEI Electronic Journal*, 10(2), 2007.

[51] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe. Developing the UML as a formal modelling notation. In *Proc. UML' 98, LNCS*, volume 1618, 1998.

[52] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000.

[53] Michael Findling. REST and SOAP: When Should I Use Each (or Both)? `http://edn.embarcadero.com/article/40558`. Accessed: 2013-11-01.

[54] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Sigplan Notices*, volume 37, pages 234–245. ACM, 2002.

[55] Ian Foster, Savas Parastatidis, Paul Watson, and Mark Mckeown. How do I model state?: Let me count the ways. *Communications of the ACM*, 51(9):34–41, 2008.

[56] William Frakes and Carol Terry. Software Reuse: Metrics and Models. *ACM Computing Surveys (CSUR)*, 28(2):415–435, 1996.

[57] Xiang Fu, Tevfik Bultan, and Jianwen Su. Synchronizability of conversations among web services. *IEEE Transactions on Software Engineering*, 31(12):1042–1055, 2005.

[58] Miguel Garcia and A Jibran Shidqie. OCL Compiler for EMF. In *Eclipse Modeling Symposium at Eclipse Summit Europe*, 2007.

[59] José García-Fanjul, Javier Tuya, and Claudio De La Riva. Generating test cases specifications for BPEL compositions of web services using SPIN. In *International Workshop on Web Services–Modeling and Testing (WS-MaTe)*, page 83, 2006.

[60] Eran Gery, David Harel, and Eldad Palachi. Rhapsody: A complete lifecycle model-based development system. In *Integrated Formal Methods*, pages 1–10. Springer, 2002.

[61] Nawal Guermouche and Claude Godart. Timed model checking based approach for web services analysis. In *IEEE International Conference on Web Services (ICWS)*, pages 213–221. IEEE, 2009.

[62] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3):231–274, 1987.

[63] Jan Hendrik Hausmann, Reiko Heckel, and Marc Lohmann. Model-based development of web services descriptions enabling a precise matching concept. *International Journal of Web Services Research (IJWSR)*, 2(2):67–84, 2005.

[64] Reiko Heckel, Hendrik Voigt, Jochen Kuster, and Sebastian Thone. Towards Consistency of Web Service Architectures. In *Proceedings of the 7th World Multiconference on Systemics, Cybernetics, and Informatics*, 2003.

[65] Anders Hessel, Kim Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. Springer-Verlag, 2008.

[66] Adrian Holovaty and Jacob Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.

[67] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[68] GJ Holzmann. Promela language reference. *Bell Labs*, 1997.

[69] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The Even More Irresistible SROIQ. *KR*, 6:57–67, 2006.

[70] Ian Horrocks, F. Peter, Patel Schneider, and Frank Van Harmelen. From $\mathscr{SHIQ}$ and RDF to OWL: The making of a web ontology language. *J. of Web Semantics*, 1(1):7–26, 2003.

[71] Hai Huang, W-T Tsai, and Raymond Paul. Automated model checking and testing for composite web services. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 300–307. IEEE, 2005.

[72] Naseem Ibrahim and Ismail Al Ani. Beyond Functional Verification of Web Services Compositions. *Journal of Emerging Trends in Computing and Information Sciences*, 4, Special Issue:25–30, 2013.

[73] Pertti Järvinen. *On research methods*. Opinpajan kirja, 2001.

[74] Sebastian Kochman, Paweł T Wojciechowski, and Miłosz Kmieciak. Batched transactions for RESTful web services. In *Current Trends in Web Engineering*, pages 86–98. Springer, 2012.

[75] Martin Koskinen, Dragos Truscan, Tanwir Ahmad, and Niklas Grönblom. Combining Model-based Testing and Continuous Integration. In *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pages 65–71, 2013.

[76] Janne Kuuskeri and Tuomas Turto. On Actors and the REST. In *Web Engineering*, pages 144–157. Springer, 2010.

[77] Markku Laitkorpi, Johannes Koskinen, and Tarja Systa. A UML-based approach for abstracting application interfaces to ReST-like services. In *13th Working Conference on Reverse Engineering (WCRE)*, pages 134–146. IEEE, 2006.

[78] Markku Laitkorpi, Petri Selonen, and Tarja Systa. Towards a model-driven process for designing restful web services. In *IEEE International Conference on Web Services(ICWS)*, pages 173–180. IEEE, 2009.

[79] Mounir Lallali, Fatiha Zaidi, Ana Cavalli, and Iksoon Hwang. Automatic timed test case generation for web services composition. In *IEEE Sixth European Conference on Web Services (ECOWS)*, pages 53–62. IEEE, 2008.

[80] Jani Lampinen. Interface specification methods for software components. *TKK Reports in Information and Computer Science*, 2008.

[81] Kim G Larsen, Marius Mikucionis, and Brian Nielsen. UPPAAL TRON user manual. *CISS, BRICS, Aalborg University, Aalborg, Denmark*, 2009.

[82] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.

[83] Gary T Leavens, Albert L Baker, and Clyde Ruby. JML: a Java modeling language. In *Formal Underpinnings of Java Workshop (OOPSLA)*, 1998.

[84] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. Reconciling manual and automated testing: The autotest experience. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261a–261a. IEEE, 2007.

[85] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract driven development= test driven development-writing test cases. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 425–434. ACM, 2007.

[86] Marc Lohmann, Leonardo Mariani, and Reiko Heckel. A model-driven approach to discovery, testing and monitoring of web services. In *Test and Analysis of Web Services*, pages 173–204. Springer, 2007.

[87] Salvatore T March and Gerald F Smith. Design and natural science research on information technology. *Decision support systems*, 15(4):251–266, 1995.

[88] Alexandros Marinos, Amir Razavi, Sotiris Moschoyiannis, and Paul Krause. RETRO: A consistent and recoverable RESTful transaction model. In *Web Services, 2009. ICWS 2009. IEEE International Conference on*, pages 181–188. IEEE, 2009.

[89] David Martin et al. OWL-S: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.

[90] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (URI): Generic syntax. 2005.

[91] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[92] Nikola Milanovic. *Contract-based web service composition*. PhD thesis, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät II, 2006.

[93] Boris Motik, Peter F Patel-Schneider, Bijan Parsia, Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, et al. OWL 2 web ontology language: Structural specification and functional-style syntax. *W3C recommendation*, 27:17, 2009.

[94] Christian Nentwich, Wolfgang Emmerich, and Anthony Finkelstein. Static consistency checking for distributed specifications. In *Proceedings of 16th Annual International Conference on Automated Software Engineering*, pages 115–124. IEEE, 2001.

[95] Iftikhar Azim Niaz and Jiro Tanaka. Mapping UML statecharts to Java Code. In *IASTED Conference on Software Engineering*, pages 111–116, 2004.

[96] OMG. *OCL, OMG Available Specification, Version 2.0*, 2006.

[97] Emilio Ormeno, Marıa Lund, Laura Aballay, and Silvana Aciar. An UML profile for modeling RESTful services. *13th Argentine Symposium on Software Engineering (ASSE)*, pages 119–133, 2012.

[98] Guy Pardon and Cesare Pautasso. Atomic distributed transactions: a RESTful design. In *Proceedings of the companion publication of the 23rd International Conference on World Wide Web companion*, pages 943–948. International World Wide Web Conferences Steering Committee, 2014.

[99] Oscar Pastor, Sergio España, José Ignacio Panach, and Nathalie Aquino. Model-driven development. *Informatik-Spektrum*, 31(5):394–407, 2008.

[100] Cesare Pautasso. BPEL for REST. *Business Process Management*, pages 278–293, 2008.

[101] Cesare Pautasso. Composing RESTful services with Jopera. *Software Composition*, pages 142–159, 2009.

[102] Cesare Pautasso. RESTful Web service composition with BPEL for REST. *Data & Knowledge Engineering*, 68(9):851–866, 2009.

[103] Tom Pender, Eugene McSheffrey, and Lou Varveris. *UML bible*. Wiley Chichester, 2003.

[104] Sandy Pérez, Frederico Durao, Santiago Meliá, Peter Dolog, and Oscar Díaz. RESTful, Resource-Oriented Architectures:A Model-Driven Approach. In *Web Information Systems Engineering–WISE 2010 Workshops*, pages 282–294. Springer, 2011.

[105] Francisco AC Pinheiro and Joseph A Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, 13(2):52–64, 1996.

[106] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: A Decidable (Yet Expressive) Fragment of OCL. In *Proceedings of the 25th Int. Workshop on Description Logics*, volume 846 of *CEUR Electronic Workshop Proceedings, http://ceur-ws.org/*, pages 312–322, 2012.

[107] Anna Queralt, Alessandro Artale, Diego Calvanese, and Ernest Teniente. OCL-Lite: Finite Reasoning on UML/OCL Conceptual Schemas. *Data and Knowledge Engineering*, 73:1–22, 2012.

[108] Shearer R, Motik B, and Horrocks I. HermiT: A highly-efficient OWL reasoner. *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, 2008.

[109] Irum Rauf, M Iqbal, and Zafar I Malik. UML based modeling of web service composition-a survey. *Sixth International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 301–307, 2008.

[110] Amir Razavi, Alexandros Marinos, Sotiris Moschoyiannis, and Paul Krause. RESTful transactions supported by the isolation theorems. In *Web Engineering*, pages 394–409. Springer, 2009.

[111] Leonard Richardson and Sam Ruby. *RESTful web services*. O'Reilly, 2008.

[112] Florian Rosenberg, Francisco Curbera, Matthew J Duftler, and Rania Khalaf. Composing Restful services and collaborative workflows: A lightweight approach. *IEEE Internet Computing*, 12(5):24–31, 2008.

[113] Anna Ruokonen, Lasse Pajunen, and Tarja Systa. Scenario-driven approach for business process modeling. *IEEE International Conference on Web Services (ICWS)*, pages 123–130, 2009.

[114] Hazlifah Mohd Rusli, Suhaimi Ibrahim, and Mazidah Puteh. Testing Web services composition: a mapping study. *Communications of the IBIMA*, 2007:34–48, 2011.

[115] Silvia Schreier. Modeling RESTful applications. In *Proceedings of the Second International Workshop on RESTful Design*, pages 15–21. ACM, 2011.

[116] Q.Z. Sheng, B. Benatallah, M. Dumas, and E.O.Y. Mak. SELF-SERV: a platform for rapid composition of web services in a peer-to-peer environment. *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 1051–1054, 2002.

[117] Mika Siikarla, Markku Laitkorpi, Petri Selonen, and Tarja Systä. Transformations have to be developed ReST assured. In *Theory and Practice of Model Transformations*, pages 1–15. Springer, 2008.

[118] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[119] B. Srivastava and J. Koehler. Web service composition-current solutions and open problems. *Proceedings of ICAPS 2003 Workshop on Planning for Web Services*, pages 28–35, 2003.

[120] Jakob Strauch and Silvia Schreier. RESTify: from RPCs to RESTful HTTP design. In *Proceedings of the Third International Workshop on RESTful Design*, pages 11–18. ACM, 2012.

[121] Toshiro Takase, Satoshi Makino, Shinya Kawanaka, Ken Ueno, Christopher Ferris, and Arthur Ryman. Definition languages for RESTful Web services: WADL vs. WSDL 2.0. *IBM Reasearch*, 2008.

[122] Wei-Tek Tsai, Xiao Wei, Yinong Chen, Bingnan Xiao, R Paul, and Hai Huang. Developing and assuring trustworthy Web services. pages 43–50, 2005.

[123] WT Tsai, X Wei, Y Chen, and R Paul. A Robust Testing Framework for Verifying Web Services by Completeness and Consistency Analysis. In

IEEE International Workshop Service-Oriented System Engineering, pages 151–158. IEEE, 2005.

[124] Dmitry Tsarkov and Ian Horrocks. FaCT++ description logic reasoner: system description. In *Proceedings of the Third International Joint Conference on Automated Reasoning*, IJCAR'06, pages 292–297, Berlin, Heidelberg, 2006. Springer-Verlag.

[125] OMG UML. 2.4. 1 superstructure specification. Technical report, document formal/2011-08-06. Technical report, OMG, 2011.

[126] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[127] Jim Webber, Savas Parastatidis, and Ian Robinson. *REST in Practice: Hypermedia and Systems Architecture*. O'Reilly Media, Inc., 2010.

[128] Stephen A White. *BPMN modeling and reference guide: understanding and using BPMN*. Future Strategies Inc., 2008.

[129] Yu Yu Yin, JianWei Yin, Ying Li, and ShuiGuang Deng. Verifying Consistency of Web Services Behavior Using Type Theory. In *Asia-Pacific Services Computing Conference, 2008. APSCC'08. IEEE*, pages 1560–1567. IEEE, 2008.

[130] Hao Yu, Cheng Zhu, Hongming Cai, and Boyi Xu. Role-centric RESTful services description and composition for e-business applications. In *IEEE International Conference on e-Business Engineering (ICEBE)*, pages 103–110. IEEE, 2009.

[131] Guangquan Zhang, Huijuan Shi, Mei Rong, and Haojun Di. Model checking for asynchronous web service composition based on XYZ/ADL. In *Web Information Systems and Mining*, pages 428–435. Springer, 2011.

[132] Haibo Zhao and Prashant Doshi. Towards Automated RESTful Web Service Composition. pages 189–196, 2009.

[133] Xia Zhao, Enjie Liu, and Gordon J Clapworthy. A Two-Stage RESTful Web Service Composition Method Based on Linear Logic. In *Ninth IEEE European Conference onWeb Services (ECOWS)*, pages 39–46. IEEE, 2011.

[134] Xue-Yang Zhu and Zhi-Song Tang. A temporal logic-based software architecture description language XYZ/ADL. *Journal of Software*, 14(4):713–720, 2003.

[135] Ivan Zuzak, Ivan Budiselic, and Goran Delac. A finite-state machine approach for modeling and analyzing restful systems. *Journal of Web Engineering*, 10(4):353–390, 2011.

[136] Ivan Zuzak and Silvia Schreier. ArRESTed Development: Guidelines for Designing REST Frameworks. *IEEE Internet Computing*, 16(4), 2012.

Turku Centre for Computer Science
TUCS Dissertations

1. **Marjo Lipponen**, On Primitive Solutions of the Post Correspondence Problem
2. **Timo Käkölä**, Dual Information Systems in Hyperknowledge Organizations
3. **Ville Leppänen**, Studies on the Realization of PRAM
4. **Cunsheng Ding**, Cryptographic Counter Generators
5. **Sami Viitanen**, Some New Global Optimization Algorithms
6. **Tapio Salakoski**, Representative Classification of Protein Structures
7. **Thomas Långbacka**, An Interactive Environment Supporting the Development of Formally Correct Programs
8. **Thomas Finne**, A Decision Support System for Improving Information Security
9. **Valeria Mihalache**, Cooperation, Communication, Control. Investigations on Grammar Systems.
10. **Marina Waldén**, Formal Reasoning About Distributed Algorithms
11. **Tero Laihonen**, Estimates on the Covering Radius When the Dual Distance is Known
12. **Lucian Ilie**, Decision Problems on Orders of Words
13. **Jukkapekka Hekanaho**, An Evolutionary Approach to Concept Learning
14. **Jouni Järvinen**, Knowledge Representation and Rough Sets
15. **Tomi Pasanen**, In-Place Algorithms for Sorting Problems
16. **Mika Johnsson**, Operational and Tactical Level Optimization in Printed Circuit Board Assembly
17. **Mats Aspnäs**, Multiprocessor Architecture and Programming: The Hathi-2 System
18. **Anna Mikhajlova**, Ensuring Correctness of Object and Component Systems
19. **Vesa Torvinen**, Construction and Evaluation of the Labour Game Method
20. **Jorma Boberg**, Cluster Analysis. A Mathematical Approach with Applications to Protein Structures
21. **Leonid Mikhajlov**, Software Reuse Mechanisms and Techniques: Safety Versus Flexibility
22. **Timo Kaukoranta**, Iterative and Hierarchical Methods for Codebook Generation in Vector Quantization
23. **Gábor Magyar**, On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems
24. **Linas Laibinis**, Mechanised Formal Reasoning About Modular Programs
25. **Shuhua Liu**, Improving Executive Support in Strategic Scanning with Software Agent Systems
26. **Jaakko Järvi**, New Techniques in Generic Programming – C++ is more Intentional than Intended
27. **Jan-Christian Lehtinen**, Reproducing Kernel Splines in the Analysis of Medical Data
28. **Martin Büchi**, Safe Language Mechanisms for Modularization and Concurrency
29. **Elena Troubitsyna**, Stepwise Development of Dependable Systems
30. **Janne Näppi**, Computer-Assisted Diagnosis of Breast Calcifications
31. **Jianming Liang**, Dynamic Chest Images Analysis
32. **Tiberiu Seceleanu**, Systematic Design of Synchronous Digital Circuits
33. **Tero Aittokallio**, Characterization and Modelling of the Cardiorespiratory System in Sleep-Disordered Breathing
34. **Ivan Porres**, Modeling and Analyzing Software Behavior in UML
35. **Mauno Rönkkö**, Stepwise Development of Hybrid Systems
36. **Jouni Smed**, Production Planning in Printed Circuit Board Assembly
37. **Vesa Halava**, The Post Correspondence Problem for Market Morphisms
38. **Ion Petre**, Commutation Problems on Sets of Words and Formal Power Series
39. **Vladimir Kvassov**, Information Technology and the Productivity of Managerial Work
40. **Frank Tétard**, Managers, Fragmentation of Working Time, and Information Systems

41. **Jan Manuch**, Defect Theorems and Infinite Words
42. **Kalle Ranto**, Z_4-Goethals Codes, Decoding and Designs
43. **Arto Lepistö**, On Relations Between Local and Global Periodicity
44. **Mika Hirvensalo**, Studies on Boolean Functions Related to Quantum Computing
45. **Pentti Virtanen**, Measuring and Improving Component-Based Software Development
46. **Adekunle Okunoye**, Knowledge Management and Global Diversity – A Framework to Support Organisations in Developing Countries
47. **Antonina Kloptchenko**, Text Mining Based on the Prototype Matching Method
48. **Juha Kivijärvi**, Optimization Methods for Clustering
49. **Rimvydas Rukšėnas**, Formal Development of Concurrent Components
50. **Dirk Nowotka**, Periodicity and Unbordered Factors of Words
51. **Attila Gyenesei**, Discovering Frequent Fuzzy Patterns in Relations of Quantitative Attributes
52. **Petteri Kaitovaara**, Packaging of IT Services – Conceptual and Empirical Studies
53. **Petri Rosendahl**, Niho Type Cross-Correlation Functions and Related Equations
54. **Péter Majlender**, A Normative Approach to Possibility Theory and Soft Decision Support
55. **Seppo Virtanen**, A Framework for Rapid Design and Evaluation of Protocol Processors
56. **Tomas Eklund**, The Self-Organizing Map in Financial Benchmarking
57. **Mikael Collan**, Giga-Investments: Modelling the Valuation of Very Large Industrial Real Investments
58. **Dag Björklund**, A Kernel Language for Unified Code Synthesis
59. **Shengnan Han**, Understanding User Adoption of Mobile Technology: Focusing on Physicians in Finland
60. **Irina Georgescu**, Rational Choice and Revealed Preference: A Fuzzy Approach
61. **Ping Yan**, Limit Cycles for Generalized Liénard-Type and Lotka-Volterra Systems
62. **Joonas Lehtinen**, Coding of Wavelet-Transformed Images
63. **Tommi Meskanen,** On the NTRU Cryptosystem
64. **Saeed Salehi**, Varieties of Tree Languages
65. **Jukka Arvo**, Efficient Algorithms for Hardware-Accelerated Shadow Computation
66. **Mika Hirvikorpi**, On the Tactical Level Production Planning in Flexible Manufacturing Systems
67. **Adrian Costea**, Computational Intelligence Methods for Quantitative Data Mining
68. **Cristina Seceleanu**, A Methodology for Constructing Correct Reactive Systems
69. **Luigia Petre**, Modeling with Action Systems
70. **Lu Yan**, Systematic Design of Ubiquitous Systems
71. **Mehran Gomari**, On the Generalization Ability of Bayesian Neural Networks
72. **Ville Harkke**, Knowledge Freedom for Medical Professionals – An Evaluation Study of a Mobile Information System for Physicians in Finland
73. **Marius Cosmin Codrea**, Pattern Analysis of Chlorophyll Fluorescence Signals
74. **Aiying Rong**, Cogeneration Planning Under the Deregulated Power Market and Emissions Trading Scheme
75. **Chihab BenMoussa**, Supporting the Sales Force through Mobile Information and Communication Technologies: Focusing on the Pharmaceutical Sales Force
76. **Jussi Salmi**, Improving Data Analysis in Proteomics
77. **Orieta Celiku**, Mechanized Reasoning for Dually-Nondeterministic and Probabilistic Programs
78. **Kaj-Mikael Björk**, Supply Chain Efficiency with Some Forest Industry Improvements
79. **Viorel Preoteasa**, Program Variables – The Core of Mechanical Reasoning about Imperative Programs
80. **Jonne Poikonen**, Absolute Value Extraction and Order Statistic Filtering for a Mixed-Mode Array Image Processor
81. **Luka Milovanov**, Agile Software Development in an Academic Environment
82. **Francisco Augusto Alcaraz Garcia**, Real Options, Default Risk and Soft Applications
83. **Kai K. Kimppa**, Problems with the Justification of Intellectual Property Rights in Relation to Software and Other Digitally Distributable Media
84. **Dragoş Truşcan**, Model Driven Development of Programmable Architectures
85. **Eugen Czeizler**, The Inverse Neighborhood Problem and Applications of Welch Sets in Automata Theory

86. **Sanna Ranto**, Identifying and Locating-Dominating Codes in Binary Hamming Spaces
87. **Tuomas Hakkarainen**, On the Computation of the Class Numbers of Real Abelian Fields
88. **Elena Czeizler**, Intricacies of Word Equations
89. **Marcus Alanen**, A Metamodeling Framework for Software Engineering
90. **Filip Ginter**, Towards Information Extraction in the Biomedical Domain: Methods and Resources
91. **Jarkko Paavola**, Signature Ensembles and Receiver Structures for Oversaturated Synchronous DS-CDMA Systems
92. **Arho Virkki**, The Human Respiratory System: Modelling, Analysis and Control
93. **Olli Luoma**, Efficient Methods for Storing and Querying XML Data with Relational Databases
94. **Dubravka Ilić**, Formal Reasoning about Dependability in Model-Driven Development
95. **Kim Solin**, Abstract Algebra of Program Refinement
96. **Tomi Westerlund**, Time Aware Modelling and Analysis of Systems-on-Chip
97. **Kalle Saari**, On the Frequency and Periodicity of Infinite Words
98. **Tomi Kärki**, Similarity Relations on Words: Relational Codes and Periods
99. **Markus M. Mäkelä**, Essays on Software Product Development: A Strategic Management Viewpoint
100. **Roope Vehkalahti**, Class Field Theoretic Methods in the Design of Lattice Signal Constellations
101. **Anne-Maria Ernvall-Hytönen**, On Short Exponential Sums Involving Fourier Coefficients of Holomorphic Cusp Forms
102. **Chang Li**, Parallelism and Complexity in Gene Assembly
103. **Tapio Pahikkala**, New Kernel Functions and Learning Methods for Text and Data Mining
104. **Denis Shestakov**, Search Interfaces on the Web: Querying and Characterizing
105. **Sampo Pyysalo**, A Dependency Parsing Approach to Biomedical Text Mining
106. **Anna Sell**, Mobile Digital Calendars in Knowledge Work
107. **Dorina Marghescu**, Evaluating Multidimensional Visualization Techniques in Data Mining Tasks
108. **Tero Säntti**, A Co-Processor Approach for Efficient Java Execution in Embedded Systems
109. **Kari Salonen**, Setup Optimization in High-Mix Surface Mount PCB Assembly
110. **Pontus Boström**, Formal Design and Verification of Systems Using Domain-Specific Languages
111. **Camilla J. Hollanti**, Order-Theoretic Mehtods for Space-Time Coding: Symmetric and Asymmetric Designs
112. **Heidi Himmanen**, On Transmission System Design for Wireless Broadcasting
113. **Sébastien Lafond**, Simulation of Embedded Systems for Energy Consumption Estimation
114. **Evgeni Tsivtsivadze**, Learning Preferences with Kernel-Based Methods
115. **Petri Salmela**, On Commutation and Conjugacy of Rational Languages and the Fixed Point Method
116. **Siamak Taati**, Conservation Laws in Cellular Automata
117. **Vladimir Rogojin**, Gene Assembly in Stichotrichous Ciliates: Elementary Operations, Parallelism and Computation
118. **Alexey Dudkov**, Chip and Signature Interleaving in DS CDMA Systems
119. **Janne Savela**, Role of Selected Spectral Attributes in the Perception of Synthetic Vowels
120. **Kristian Nybom**, Low-Density Parity-Check Codes for Wireless Datacast Networks
121. **Johanna Tuominen**, Formal Power Analysis of Systems-on-Chip
122. **Teijo Lehtonen**, On Fault Tolerance Methods for Networks-on-Chip
123. **Eeva Suvitie**, On Inner Products Involving Holomorphic Cusp Forms and Maass Forms
124. **Linda Mannila**, Teaching Mathematics and Programming – New Approaches with Empirical Evaluation
125. **Hanna Suominen**, Machine Learning and Clinical Text: Supporting Health Information Flow
126. **Tuomo Saarni**, Segmental Durations of Speech
127. **Johannes Eriksson**, Tool-Supported Invariant-Based Programming

128. **Tero Jokela**, Design and Analysis of Forward Error Control Coding and Signaling for Guaranteeing QoS in Wireless Broadcast Systems
129. **Ville Lukkarila**, On Undecidable Dynamical Properties of Reversible One-Dimensional Cellular Automata
130. **Qaisar Ahmad Malik**, Combining Model-Based Testing and Stepwise Formal Development
131. **Mikko-Jussi Laakso**, Promoting Programming Learning: Engagement, Automatic Assessment with Immediate Feedback in Visualizations
132. **Riikka Vuokko**, A Practice Perspective on Organizational Implementation of Information Technology
133. **Jeanette Heidenberg**, Towards Increased Productivity and Quality in Software Development Using Agile, Lean and Collaborative Approaches
134. **Yong Liu**, Solving the Puzzle of Mobile Learning Adoption
135. **Stina Ojala**, Towards an Integrative Information Society: Studies on Individuality in Speech and Sign
136. **Matteo Brunelli**, Some Advances in Mathematical Models for Preference Relations
137. **Ville Junnila**, On Identifying and Locating-Dominating Codes
138. **Andrzej Mizera**, Methods for Construction and Analysis of Computational Models in Systems Biology. Applications to the Modelling of the Heat Shock Response and the Self-Assembly of Intermediate Filaments.
139. **Csaba Ráduly-Baka**, Algorithmic Solutions for Combinatorial Problems in Resource Management of Manufacturing Environments
140. **Jari Kyngäs**, Solving Challenging Real-World Scheduling Problems
141. **Arho Suominen**, Notes on Emerging Technologies
142. **József Mezei**, A Quantitative View on Fuzzy Numbers
143. **Marta Olszewska**, On the Impact of Rigorous Approaches on the Quality of Development
144. **Antti Airola**, Kernel-Based Ranking: Methods for Learning and Performace Estimation
145. **Aleksi Saarela**, Word Equations and Related Topics: Independence, Decidability and Characterizations
146. **Lasse Bergroth**, Kahden merkkijonon pisimmän yhteisen alijonon ongelma ja sen ratkaiseminen
147. **Thomas Canhao Xu**, Hardware/Software Co-Design for Multicore Architectures
148. **Tuomas Mäkilä**, Software Development Process Modeling – Developers Perspective to Contemporary Modeling Techniques
149. **Shahrokh Nikou**, Opening the Black-Box of IT Artifacts: Looking into Mobile Service Characteristics and Individual Perception
150. **Alessandro Buoni**, Fraud Detection in the Banking Sector: A Multi-Agent Approach
151. **Mats Neovius**, Trustworthy Context Dependency in Ubiquitous Systems
152. **Fredrik Degerlund**, Scheduling of Guarded Command Based Models
153. **Amir-Mohammad Rahmani-Sane**, Exploration and Design of Power-Efficient Networked Many-Core Systems
154. **Ville Rantala**, On Dynamic Monitoring Methods for Networks-on-Chip
155. **Mikko Pelto**, On Identifying and Locating-Dominating Codes in the Infinite King Grid
156. **Anton Tarasyuk**, Formal Development and Quantitative Verification of Dependable Systems
157. **Muhammad Mohsin Saleemi**, Towards Combining Interactive Mobile TV and Smart Spaces: Architectures, Tools and Application Development
158. **Tommi J. M. Lehtinen**, Numbers and Languages
159. **Peter Sarlin**, Mapping Financial Stability
160. **Alexander Wei Yin**, On Energy Efficient Computing Platforms
161. **Mikołaj Olszewski**, Scaling Up Stepwise Feature Introduction to Construction of Large Software Systems
162. **Maryam Kamali**, Reusable Formal Architectures for Networked Systems
163. **Zhiyuan Yao**, Visual Customer Segmentation and Behavior Analysis – A SOM-Based Approach
164. **Timo Jolivet**, Combinatorics of Pisot Substitutions
165. **Rajeev Kumar Kanth**, Analysis and Life Cycle Assessment of Printed Antennas for Sustainable Wireless Systems
166. **Khalid Latif**, Design Space Exploration for MPSoC Architectures

167. **Bo Yang**, Towards Optimal Application Mapping for Energy-Efficient Many-Core Platforms
168. **Ali Hanzala Khan**, Consistency of UML Based Designs Using Ontology Reasoners
169. **Sonja Leskinen**, m-Equine: IS Support for the Horse Industry
170. **Fareed Ahmed Jokhio**, Video Transcoding in a Distributed Cloud Computing Environment
171. **Moazzam Fareed Niazi**, A Model-Based Development and Verification Framework for Distributed System-on-Chip Architecture
172. **Mari Huova**, Combinatorics on Words: New Aspects on Avoidability, Defect Effect, Equations and Palindromes
173. **Ville Timonen**, Scalable Algorithms for Height Field Illumination
174. **Henri Korvela**, Virtual Communities – A Virtual Treasure Trove for End-User Developers
175. **Kameswar Rao Vaddina**, Thermal-Aware Networked Many-Core Systems
176. **Janne Lahtiranta**, New and Emerging Challenges of the ICT-Mediated Health and Well-Being Services
177. **Irum Rauf**, Design and Validation of Stateful Composite RESTful Web Services

Turku Centre *for* Computer Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www. tucs.fi

University of Turku
Faculty of Mathematics and Natural Sciences
- Department of Information Technology
- Department of Mathematics and Statistics

Turku School of Economics
- Institute of Information Systems Science

Åbo Akademi University
Division for Natural Sciences and Technology
- Department of Information Technologies

ISBN 978-952-12-3070-7
ISSN 1239-1883

Irum Rauf

Design and Validation of Stateful Composite RESTful Web Services