

UNIVERSITY OF CRETE  
FACULTY OF SCIENCES AND ENGINEERING  
DEPARTMENT OF COMPUTER SCIENCE

# **A Novel Specification and Composition Language for Services**

by

George Baryannis

Dissertation submitted in partial fulfillment  
of the requirements for the degree of

Doctor of Philosophy

Heraklion, July 2014



UNIVERSITY OF CRETE  
DEPARTMENT OF COMPUTER SCIENCE

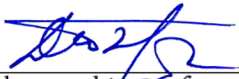
**A Novel Specification and Composition Language for Services**


Dissertation submitted by

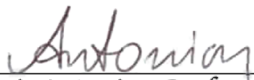
**George Baryannis**

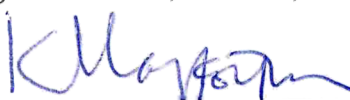
in partial fulfillment of the requirements for the  
degree of Doctor of Philosophy in Computer Science

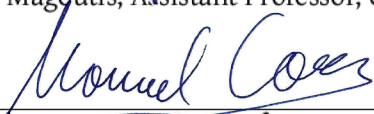
Author:   
George Baryannis, University of Crete


Examination  
Committee:   
Dimitris Plexousakis, Professor, University of Crete

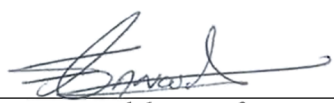
  
Christos Nikolaou, Professor, University of Crete

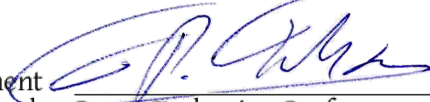
  
Grigoris Antoniou, Professor, University of Huddersfield

  
Kostas Magoutis, Assistant Professor, University of Ioannina

  
Manuel Carre, Associate Professor, Technical University of Madrid

  
Salima Benbernou, Professor, Paris Descartes University

  
George Spanoudakis, Professor, City University London

Department  
approval:   
Panos Trahanias, Professor and Department Chair

Heraklion, July 2014



# Abstract

Service-Oriented Architecture (SOA) has emerged as a prominent design style that enables an IT infrastructure to allow different applications to participate in business processes, regardless of their underlying features, by encapsulating them as platform-independent entities that become available via a certain network, primarily the Internet. In order to effectively discover and use the most suitable services, service description should provide a complete behavior model, describing the inputs and preconditions that are required before execution, as well as the outputs and effects of a successful execution. Such service specifications are indispensable in a variety of activities, such as conformance and verification checks, adaptation evaluation and deducing composability of services.

Service specifications rely on the expression of conditions that should hold before and after service execution. Such specifications are prone to a family of problems, known in the AI literature as the frame, ramification and qualification problems. These problems deal with the succinct and flexible representation of non-effects, indirect effects and preconditions, respectively. Research in services has largely ignored these problems, at the same time ignoring their effects, such as compromising the integrity and correctness of services and service compositions and the inability to provide justification for unexpected execution results.

To address these issues, this thesis proposes the Web Service Specification Language (WSSL), a novel, semantics-aware language for the specification and composition of services, independent of service design models. WSSL's foundation is the fluent calculus, a specification language for robots that offers solutions

to the frame, ramification and qualification problems. Further language extensions achieve three major goals: realize service composition via planning, supporting non-deterministic constructs, such as conditionals and loops; include specification of QoS profiles; and support partially observable service states.

To investigate WSSL's applicability and demonstrate its benefits, we analyze correctness of the composition extension, decidability and complexity of the underlying theory, as well as compatibility with other related languages in service science. Moreover, an innovative service composition and verification framework is implemented, that advances state-of-the-art by satisfying several desirable requirements simultaneously: ramifications and partial observability in service and goal modeling; non-determinism in composition schemas; dynamic binding of tasks to concrete services; explanations for unexpected behavior; QoS-awareness through pruning and ranking techniques based on heuristics and task-specific goals and an all-encompassing QoS aggregation method for global goals.

Experimental evaluation is performed using synthetically generated specifications and composition goals, investigating performance scalability in terms of execution time, as well as optimality with regard to the produced composite process. The results show that, even in the presence of ramifications in some specifications, functional planning is efficient for repositories up to 500 specifications. Also, the cost of functional discovery per single service is insignificant, hence achieving good performance even when executed for multiple candidate plans. Finally, optimality relies mainly on defining suitable problem-specific heuristics; thus, its success depends mostly on the expertise of the composition designer.

**Keywords:** Service Description, Formal Specification, Service Composition, Frame Problem, Ramification Problem, Qualification Problem, Service Verification

**Supervisor:** Dimitris Plexousakis

Professor

Computer Science Department

University of Crete

# Περίληψη

Η Υπηρεσιοστρεφής Αρχιτεκτονική έχει αναδειχθεί σε εξέχοντα τρόπο σχεδίασης που καθιστά τις υποδομές Πληροφοριακής Τεχνολογίας ικανές να παρέχουν σε εφαρμογές τη δυνατότητα ανταλλαγής δεδομένων και συμμετοχής σε επιχειρησιακές διεργασίες, ανεξάρτητα από τα υποκείμενα χαρακτηριστικά τους, όπως την επακριβή υλοποίηση ή τα λειτουργικά συστήματα και τις γλώσσες προγραμματισμού που χρησιμοποιήθηκαν για την ανάπτυξή τους. Με αυτό τον τρόπο οι υπηρεσίες χρησιμοποιούνται ως οντότητες ανεξάρτητες από πλατφόρμες, που παρέχουν πρόσβαση σε σύνολα λειτουργικότητας μέσω προκαθορισμένων διεπαφών και διατίθενται μέσω δικτύων, πρωτίστως μέσω Διαδικτύου.

Για την αποτελεσματική εύρεση και χρήση των πιο κατάλληλων υπηρεσιών (ή συνθέσεων υπηρεσιών) αναφορικά με τις ανάγκες ενός καταναλωτή, ο πάροχος θα πρέπει να προσφέρει πλήρεις προδιαγραφές για τις υπηρεσίες, που δεν περιορίζονται στις παρεχόμενες διεπαφές (υπό τη μορφή συνόλων από εισόδους και εξόδους). Αντίθετα, είναι αναγκαίο ένα πλήρες μοντέλο συμπεριφοράς που θα περιγράφει τις εισόδους και συνθήκες που απαιτούνται πριν την εκτέλεση, καθώς επίσης και τις εξόδους και αποτελέσματα που προκύπτουν από μια επιτυχή εκτέλεση. Τέτοιες προδιαγραφές υπηρεσιών είναι απαραίτητες σε μια πληθώρα από δραστηριότητες όπως την κατασκευή μιας υπηρεσίας βάσει προδιαγραφών, ελέγχους συμμόρφωσης και επαλήθευσης σωστής λειτουργίας, αξιολόγηση αποτελεσμάτων της διαδικασίας προσαρμογής υπηρεσιών και τον προσδιορισμό της δυνατότητας σύνθεσης ενός συνόλου υπηρεσιών.

Οι προδιαγραφές υπηρεσιών βασίζονται στην έκφραση συνθηκών που πρέπει

να ισχύουν πριν και μετά την εκτέλεση της υπηρεσίας. Τέτοιες προδιαγραφές είναι επιρρεπείς σε μια οικογένεια προβλημάτων, που είναι γνωστά στο πεδίο της Τεχνητής Νοημοσύνης ως το πρόβλημα πλαισίου, το πρόβλημα επιπτώσεων και το πρόβλημα προϋποθέσεων. Το πρόβλημα πλαισίου αφορά την έκφραση των μη-αποτελεσμάτων μιας ενέργειας, δηλαδή την περιγραφή του τι παραμένει αμετάβλητο μετά την εκτέλεση. Το πρόβλημα επιπτώσεων ασχολείται με τη μοντελοποίηση των έμμεσων αποτελεσμάτων ή δευτερευουσών συνεπειών, δηλαδή των συνεπειών που ακολουθούν ένα πρωταρχικό αποτέλεσμα. Τέλος, το πρόβλημα προϋποθέσεων, σε αντίθεση με τα άλλα δύο, σχετίζεται με την αναπαράσταση των προϋποθέσεων εκτέλεσης και την αδυναμία του να ληφθούν υπόψη όλες οι δυνατές συνθήκες κάτω από οποιαδήποτε περίσταση.

Ενώ η έρευνα σε άλλα πεδία όπως οι προγραμματιστικές προδιαγραφές ή η συλλογιστική ενεργειών και αλλαγής έχει οδηγήσει σε ικανοποιητικές λύσεις στα προαναφερθέντα προβλήματα, η έρευνα στις υπηρεσίες τα έχει αγνοήσει σε μεγάλο βαθμό, αγνοώντας ταυτόχρονα και τις συνέπειές τους. Πιο συγκεκριμένα, η αποτυχία επίλυσης του προβλήματος πλαισίου οδηγεί σε περιγραφές υπηρεσιών που δε μπορούν να εγγυηθούν στους καταναλωτές υπηρεσιών ότι τα καθορισμένα αποτελέσματα είναι τα μόνα που απορρέουν από μια εκτέλεση. Το γεγονός αυτό εγείρει σημαντικά ζητήματα όπως η ακεραιότητα της παρεχόμενης υπηρεσίας και η αξιοπιστία του παρόχου, ασφάλεια και ιδιωτικότητα των εμπλεκόμενων χρηστών και δεδομένων, ενώ ταυτόχρονα διακινδυνεύεται η δυνατότητα επαναχρησιμοποίησης σε συνθέσεις υπηρεσιών. Επιπλέον, η παράβλεψη των δευτερευουσών συνεπειών οδηγεί σε ελλιπή μοντέλα συμπεριφοράς που μπορεί να οδηγήσουν σε εσφαλμένες συνθέσεις υπηρεσιών, ενώ η απουσία λύσης στο πρόβλημα προϋποθέσεων σημαίνει ταυτόχρονα και την αδυναμία αιτιολόγησης στην περίπτωση που η υπηρεσία δεν παράγει τα αναμενόμενα αποτελέσματα ενώ έχει δοθεί η απαιτούμενη πληροφορία εισόδου και έχουν ικανοποιηθεί οι προϋποθέσεις εκτέλεσης.

Για την αντιμετώπιση όλων των ζητημάτων που αναφέρθηκαν προηγουμένως, η παρούσα διατριβή προτείνει τη Γλώσσα Προδιαγραφών Υπηρεσιών (WSSL), μια νέα γλώσσα για προδιαγραφές και σύνθεση υπηρεσιών. Η θεμελίωση της γλώσ-



σας βασίζεται στο λογισμό των μεταβλητών ιδιοτήτων (fluent calculus), που συνιστά μια γλώσσα προδιαγραφών και ένα σύστημα ελέγχου αυτόνομων ρομποτικών πρακτόρων που παρέχει λύσεις στα προβλήματα πλαισίου, επιπτώσεων και προϋποθέσεων. Η WSSL είναι ανεξάρτητη από μοντέλα σχεδίασης υπηρεσιών ενώ παρέχει και τη δυνατότητα σημασιολογικού σχολιασμού. Η υποστήριξη των λύσεων στα προαναφερθέντα προβλήματα γίνεται με φυσικό και άμεσο τρόπο, με δομές που είναι ενσωματωμένες στη γλώσσα. Επιπλέον, η θεμελίωση της γλώσσας παρέχει τη δυνατότητα μετατροπής των προδιαγραφών σε προτάσεις λογικού προγραμματισμού.

Εκτός από το βασικό συντακτικό και τη σημασιολογία της γλώσσας, παρέχονται επεκτάσεις σε τρεις κατευθύνσεις για την επίτευξη τριών μείζονων στόχων. Πρώτον, υποστηρίζεται προδιαγραφή και υλοποίηση σύνθεσης υπηρεσιών, λαμβάνοντας υπόψη όλα τα θεμελιώδη πρότυπα ροής εργασίας, συμπεριλαμβανομένων μη-ντετερμινιστικών προτύπων, όπως η εκτέλεση υπό συνθήκη και η επαναληπτική εκτέλεση. Δεύτερον, δίνεται η δυνατότητα προδιαγραφής προφίλ Ποιότητας Υπηρεσιών (ΠΥ) σύμφωνα με διαθέσιμα μοντέλα και μεταμοντέλα περιγραφής ιδιοτήτων ποιότητας όπως χρόνος εκτέλεσης ή κόστος. Τρίτον, υποστηρίζεται αβεβαιότητα στην περιγραφή των καταστάσεων που συνθέτουν το μοντέλο συμπεριφοράς της υπηρεσίας, όπως για παράδειγμα, η υποστήριξη μερικώς παρατηρήσιμων αρχικών καταστάσεων στη διαδικασία σύνθεσης.

Για να διαπιστωθεί η χρησιμότητα της γλώσσας, διεξήχθη ενδελεχής ανάλυση μιας σειράς από ενδιαφέρουσες ιδιότητες. Αρχικά μελετήθηκε η ορθότητα της επέκτασης που αφορά τη σύνθεση υπηρεσιών ώστε να διαπιστωθεί ότι είναι σύμφωνη με τη θεμελίωση της γλώσσας. Στη συνέχεια, αναλύθηκε η πολυπλοκότητα και αποκρισιμότητα της υποκείμενης θεωρίας λογισμού μεταβλητών ιδιοτήτων. Επίσης, εξετάστηκε η δυνατότητα εφαρμογής της γλώσσας όσον αφορά τη συμβατότητα και τη σύνδεση της με εναλλακτικές, σχετιζόμενες ή συμπληρωματικές γλώσσες στην επιστήμη υπηρεσιών, όπως οι WSDL, OWL-S, WSMO, USDL και BPMN.

Εξάλλου, για να αποδειχθούν τα σημαντικά οφέλη της WSSL, σχεδιάστηκε και υλοποιήθηκε ένα καινοτόμο σύστημα σύνθεσης και επαλήθευσης υπηρεσιών βά-

σει προδιαγραφών. Το σύστημα προάγει την έρευνα στο πεδίο της σύνθεσης υπηρεσιών ικανοποιώντας ταυτόχρονα μια σειρά από σημαντικές απαιτήσεις, χάρη στις δυνατότητες της WSSL. Ειδικότερα, υποστηρίζεται η περιγραφή δευτερευουσών συνεπειών τόσο στις υπηρεσίες όσο και στους στόχους σύνθεσης, ενώ η διαδικασία επαλήθευσης βάσει προδιαγραφών μπορεί να παρέχει επεξηγήσεις σε περίπτωση μη αναμενόμενης συμπεριφοράς, χάρη στις λύσεις στα προβλήματα επιπτώσεων και προϋποθέσεων. Επιπλέον, η διαδικασία σύνθεσης είναι αυτοματοποιημένη και παράγει σύνθετες υπηρεσίες υπό τη μορφή εργασιών ομαδοποιημένων σε ροές εργασίας, με προδιαγραφές για κάθε εργασία, επιτρέποντας τη δυναμική αντιστοίχιση συγκεκριμένων υπηρεσιών για κάθε εργασία.

Επιπλέον απαιτήσεις ικανοποιούνται χάρη στις επεκτάσεις της WSSL. Συγκεκριμένα, η σύνθεση υποστηρίζει μη-ντετερμινιστικές δομές ελέγχου, όπως η εκτέλεση βάσει συνθήκης και η επαναληπτική εκτέλεση. Επίσης, υποστηρίζεται η περιγραφή καταστάσεων που είναι μερικώς παρατηρήσιμες (είτε πριν είτε μετά την εκτέλεση μιας υπηρεσίας που συμμετέχει σε μια σύνθεση), όπως για παράδειγμα η εύρεση σύνθεσης που ικανοποιεί ένα στόχο βάσει ελλιπούς αρχικής κατάστασης. Ακόμα, υποστηρίζονται τόσο λειτουργικοί στόχοι όσο και στόχοι που αφορούν ΠΥ. Για την επιλογή βέλτιστων πλάνων εκτέλεσης και την ιεράρχηση τους βάσει ιδιοτήτων ποιότητας, χρησιμοποιούνται ευρετικοί κανόνες που αφορούν το εκάστοτε πρόβλημα σύνθεσης και στόχοι που αναφέρονται σε ποιοτικά χαρακτηριστικά μεμονωμένων εργασιών στο πλάνο. Για την επίτευξη στόχων που αφορούν χαρακτηριστικά ποιότητας ολόκληρης της σύνθεσης, παρέχεται μια καθολική μέθοδος συνάθροισης ΠΥ, βάσει μιας ολοκληρωμένης ταξινόμησης ιδιοτήτων ΠΥ με γνώμονα τη φύση και τους τύπους τιμών κάθε ιδιότητας.

Τέλος, σχεδιάστηκε και εκτελέστηκε μια εκτενής πειραματική αξιολόγηση του συστήματος, εστιάζοντας σε κάθε ξεχωριστή υπομονάδα που περιέχεται, καθώς επίσης και μια αξιολόγηση του συστήματος συνολικά. Τα πειράματα βασίζονται σε συνθετικά παραγόμενες προδιαγραφές και προβλήματα σύνθεσης που καλύπτουν ένα ευρύ φάσμα πολυπλοκότητας. Η βασικότερη μεταβλητή αξιολόγησης είναι ο χρόνος εκτέλεσης ώστε να διαπιστωθεί η δυνατότητα κλιμακωτής απόδοσης. Σε

ορισμένες περιπτώσεις αξιολογήθηκε και η κατανάλωση μνήμης, ενώ για την περίπτωση της υπομονάδας που επιλέγει και ιεραρχεί τα διάφορα πλάνα εκτέλεσης κρίθηκε και το κατά πόσο το τελικό αποτέλεσμα είναι βέλτιστο.

Τα αποτελέσματα της αξιολόγησης δείχνουν ότι, ακόμα και υπό την παρουσία δευτερευουσών συνεπειών σε ένα υποσύνολο των προδιαγραφών που λαμβάνονται υπόψη, η διαδικασία εύρεσης συνθέσεων βάσει λειτουργικών στόχων είναι αποδοτική για αποθετήρια που περιέχουν μέχρι 500 διαφορετικά έγγραφα προδιαγραφών. Επιπλέον, το κόστος της διαδικασίας εύρεσης υπηρεσιών βάσει προδιαγραφών λειτουργικότητας είναι ασήμαντο όταν πρόκειται για μία εκτέλεση (δηλαδή την εύρεση όλων των προδιαγραφών και κατ' επέκταση όλων των υπηρεσιών που ταιριάζουν με την προδιαγραφή μιας υπηρεσίας). Το γεγονός αυτό επιτρέπει τη διατήρηση της αποδοτικότητας του συστήματος, ακόμα και όταν η διαδικασία εύρεσης εκτελείται για όλες τις υπηρεσίες που περιέχονται σε πολλαπλές υποψήφιες συνθέσεις. Τέλος, η εύρεση του βέλτιστου πλάνου εκτέλεσης σχετίζεται σε μεγάλο βαθμό από τους ευρετικούς κανόνες που ορίζονται για το εκάστοτε πρόβλημα σύνθεσης και άρα εξαρτάται κυρίως από την πραγματογνωμοσύνη του σχεδιαστή συνθέσεων υπηρεσιών.

Λέξεις-κλειδιά: Περιγραφή Υπηρεσιών, Τυπικές Προδιαγραφές, Σύνθεση Υπηρεσιών, Πρόβλημα Πλαισίου, Πρόβλημα Επιπτώσεων, Πρόβλημα Προϋποθέσεων

Επόπτης Καθηγητής: Δημήτρης Πλεξουσάκης

Καθηγητής

Τμήμα Επιστήμης Υπολογιστών

Πανεπιστήμιο Κρήτης



*To the (benevolent) Men Behind The Curtain*



# Acknowledgements

The process of obtaining a PhD degree turned out to be a long and hard one, in accordance with all expectations, observations and supporting evidence. Fortunately enough, there were a lot of people along this journey that managed to make it feel somewhat shorter and easier, and certainly more fun. This thesis would not have ended up the way it did without each one of the following people, to a lesser or greater degree and, for that, they have my most sincere and deep gratitude.

First and foremost, I would like to thank my supervisor, Professor Dimitris Plexousakis, for more than 7 years of efficient and fruitful collaboration that laid the foundations of my journey in academia. His support, encouragement and expertise were the driving force behind both my MSc and PhD degrees.

I would also like to thank Professor Christos Nikolaou, member of my Advisory Committee, for our effective partnership in research projects and his valuable comments at key stages of my doctoral research. Last but not least, I wish to thank Professor Grigoris Antoniou, member of my Advisory Committee, for our collaboration during his tenure as head of the Information Systems Laboratory at FORTH and his support.

I would especially like to extend my deep gratitude to Associate Professor Manuel Carro, member of my Examination Committee, for our close collaboration and his guidance during the first years of my PhD, that had a profound impact in shaping up this thesis, for hosting me for two research visits at IMDEA and Universidad Politecnica de Madrid and for his helpful comments during and after the PhD defense.

Special thanks go to Assistant Professor Kostas Magoutis for our research collaboration at FORTH that influenced some aspects of this thesis and for his useful comments as a member of my Examination Committee, as well as Kyriakos Kritikos for his invaluable assistance and guidance at the later stages of my doctoral research. Finally, I would like to thank Professors Salima Benbernou and George Spanoudakis, members of my Examination committee, for taking the time to study my PhD thesis and provide their expertise in order to improve the final document.

I would be remiss if I did not acknowledge the financial support of the Institute of Computer Science at FORTH and the University of Crete, which played a fundamental role in providing me with an excellent environment for working and studying, as well as the opportunity to travel for the purposes of my research. Particularly, I would like to thank Maria Moutsaki for her continuous support and her quick solutions to absolutely anything I encountered during my years at FORTH, as well as her warm and fun attitude.

Undoubtedly, I owe a great deal of gratitude to all the colleagues and friends that supported me throughout this journey. Specifically, I would like to thank Chrysostomos Zeginis, Panagiotis Papadakis, Ioannis Chrysakis, Dimitra Zografistou and Roula Avgoustaki for contributing to a unique and fun working environment, as well as providing me with support and relaxation when it was most necessary. Moreover, I wish to thank Giannis Androulakis and Katerina Boutsika for their continuous and uplifting support and the fun times we spent together and Maria Michou for being the best work neighbour I ever had and for the endless discussions we had for our most favourite topic, television series and movies.

I also want to deeply thank Christina and Emmanouela Lionoudaki for all the precious moments we spent together both in Chania and Heraklion and which I will cherish forever; I also wish them lots of happiness and strength in their new roles as mom and aunt, which they assumed with the arrival of a 3800g bundle of joy on the day of my PhD defense. Furthermore, I would like to offer my most sincere gratitude to Maria Kalaitzaki for being my most trusted friend, ever since



that fateful (and extremely informative) bus ride from FORTH, and for everything that followed afterwards that I will always remember fondly.

Additionally, I would like to wholeheartedly thank Manos Papadakis for single-handedly making this past year the most unique, unexpected and wonderful I have ever experienced. His remarkable worldview, in many cases fundamentally different than mine, turned out to be exactly what I needed beside me in order to get through the homestretch and reach the finish line, with the unanticipated consequence of (hopefully) gaining a lifelong friend.

Finally, I wish to thank from the bottom of my heart my close friend Giorgos Mademlis who has been continually and unconditionally there for me for the past 11 years, through thick and thin, even when he was not physically by my side. His significant influence in my eventual decision to pursue a doctoral degree and his crucial support from start to finish earn him deservedly part of the praise.

The greatest gratitude, however, is rightfully owed to my parents, Vassili and Marianna, for their endless, selfless and invaluable love and support throughout the years and for being my one true constant in life. Mom and Dad, I could not have done this without you, so thank you both, right from the heart.



*It only ends once. Anything that happens before that is just progress.*

*– Jacob, The Incident, Part 1*



# Contents

	<b>Page</b>
1 Introduction . . . . .	1
1.1 Motivation . . . . .	3
1.1.1 Service Specifications . . . . .	3
1.1.2 Representation Problems . . . . .	5
1.1.3 Service Composition . . . . .	6
1.2 Thesis Contribution and Impact . . . . .	7
1.3 Dissertation Outline . . . . .	11
2 Background and Literature Review . . . . .	13
2.1 Running Example . . . . .	15
2.2 The Frame, Ramification and Qualification Problems . . . . .	18
2.2.1 Definitions . . . . .	18
2.2.2 The Temporal Action Logic case . . . . .	21
2.2.3 Modular-E: The Event Calculus case . . . . .	22
2.2.4 The Fluent Calculus case . . . . .	23
2.2.5 Comparison . . . . .	24
2.3 Service Description and Specification . . . . .	25
2.3.1 Web Services Description Language (WSDL) . . . . .	25
2.3.2 Semantic Annotations for WSDL (SAWSDL) . . . . .	27
2.3.3 OWL-S: Semantic Markup for Web Services . . . . .	28
2.3.4 Web Service Modeling Ontology (WSMO) . . . . .	30
2.3.5 Semantic Web Services Framework (SWSF) . . . . .	32

2.3.6	The SAVVY-WS Framework . . . . .	33
2.3.7	Unified Service Description Language (USDL) . . . . .	34
2.3.8	Conclusion . . . . .	35
2.4	Automated Service Composition . . . . .	36
2.4.1	Requirements . . . . .	37
2.4.2	Workflow-based Approaches . . . . .	43
2.4.3	Model-based Approaches . . . . .	46
2.4.4	Logic-based Approaches . . . . .	47
2.4.5	AI Planning Approaches . . . . .	49
2.4.6	Comparison and Research Challenges . . . . .	56
2.5	Specification-based Service Discovery . . . . .	61
2.5.1	Discussion . . . . .	64
2.6	Verification of Service Behavior . . . . .	65
2.6.1	Discussion . . . . .	67
3	Web Service Specification Language (WSSL) . . . . .	69
3.1	The Fluent Calculus . . . . .	71
3.1.1	General Notational Conventions . . . . .	71
3.1.2	Basic Definitions . . . . .	71
3.1.3	Actions, State Change and the Frame Problem . . . . .	73
3.1.4	Representing Inputs and Outputs . . . . .	76
3.2	Abstract Syntax . . . . .	77
3.2.1	Identifiers and Namespaces . . . . .	77
3.2.2	Service Specifications . . . . .	78
3.2.3	Addressing the Ramification Problem . . . . .	82
3.2.4	Addressing the Qualification Problem . . . . .	84
3.2.5	WSSL specification of the running example . . . . .	86
3.3	Surface Syntax . . . . .	89
3.4	Semantics . . . . .	90
3.4.1	Satisfaction and Entailment . . . . .	92
3.5	WSSL/XML . . . . .	93

4	WSSL Extensions . . . . .	97
4.1	Composition . . . . .	98
4.1.1	Calculating composite preconditions and postconditions	98
4.1.2	WSSL for Composition . . . . .	103
4.2	Quality of Service . . . . .	105
4.2.1	WSSL QoS Profiles . . . . .	106
4.2.2	Correctness of QoS Profiles . . . . .	108
4.2.3	Local QoS Goal Matching . . . . .	109
4.2.4	QoS Metrics Analysis and Aggregation . . . . .	110
4.3	Partial Observability . . . . .	113
4.3.1	Incomplete States . . . . .	115
4.3.2	Knowledge States . . . . .	116
5	Implementation . . . . .	119
5.1	Implementing WSSL in FLUX . . . . .	120
5.1.1	FLUX kernels . . . . .	120
5.1.2	Domain Encoding . . . . .	122
5.1.3	WSSL Verification Tool . . . . .	124
5.1.4	Planning . . . . .	125
5.2	WSSL/CVF: Composition and Verification Framework . . . . .	131
5.2.1	Functional Composition and Verification . . . . .	134
5.2.2	Specification-based Functional Discovery . . . . .	136
5.2.3	Extended Plan Pruning and Ranking . . . . .	138
5.2.4	QoS-based Selection . . . . .	140
5.2.5	Framework Use . . . . .	143
5.2.6	Limitations . . . . .	144
6	Property Analysis . . . . .	147
6.1	Correctness . . . . .	148
6.1.1	Holds . . . . .	148
6.1.2	Minus, Plus and Update . . . . .	149
6.1.3	Causes and Ramify . . . . .	150

6.1.4	Control Flow . . . . .	150
6.1.5	Complete FLUX Kernel . . . . .	150
6.2	Decidability and Complexity . . . . .	151
6.2.1	Core WSSL . . . . .	151
6.2.2	WSSL Extensions . . . . .	153
6.3	Applicability and Practical Concerns . . . . .	154
6.3.1	WSSL Grounding to WSDL . . . . .	155
6.3.2	Generating WSSL Specifications . . . . .	160
6.3.3	Generating BPMN from WSSL plans . . . . .	164
6.3.4	Integrating WSSL into USDL . . . . .	165
6.4	Conclusions . . . . .	165
7	Evaluation . . . . .	167
7.1	Evaluation of individual components . . . . .	169
7.1.1	WSSL-to-FLUX Translation . . . . .	169
7.1.2	WSSL Planning . . . . .	170
7.1.3	Specification-based Functional Discovery . . . . .	179
7.1.4	Extended Plan Pruning . . . . .	182
7.1.5	Extended Plan Ranking . . . . .	184
7.1.6	QoS Aggregation . . . . .	187
7.1.7	Conclusion . . . . .	187
7.2	Overall Evaluation . . . . .	190
8	Conclusions and Future Research . . . . .	195
8.1	Synopsis of Contributions . . . . .	195
8.2	Directions for Future Research . . . . .	197
	Bibliography . . . . .	217
A	WSSL BNF grammar . . . . .	219
B	WSSL XML Schema . . . . .	223
B.1	XML Schema . . . . .	223
B.2	XML Encoding of running example tasks . . . . .	230
B.2.1	ReceiveCall . . . . .	230



B.2.2	MReport . . . . .	231
B.2.3	RetrieveDiag . . . . .	233
C	FLUX Kernels . . . . .	237
C.1	Copyright Notice . . . . .	237
C.2	Basic Kernel . . . . .	238
C.3	Full Kernel . . . . .	242
C.4	FLUX code of the running example . . . . .	247



# List of Figures

2.1	Composite process of the running example . . . . .	17
2.2	Top level of the OWL-S ontology [Martin et al. 2004] . . . . .	29
2.3	The four main WSMO components [Roman et al. 2005] . . . . .	30
2.4	The PAWS framework [Ardagna et al. 2007] . . . . .	44
2.5	Composition algorithm of [Lécué et al. 2008b] . . . . .	49
2.6	Composition architecture of [Hatzi et al. 2012] . . . . .	52
2.7	Composition architecture of [Wu et al. 2007] . . . . .	53
4.1	Fundamental control constructs for service composition . . . . .	99
5.1	Overview of WSSL/CVF . . . . .	134
6.1	Placement of WSSL with regard to current and former service de- scription efforts . . . . .	154
7.1	Scalability evaluation of translation process: Time . . . . .	171
7.2	Scalability evaluation of translation process: Memory consumption	171
7.3	Building blocks of the experiment plans . . . . .	173
7.4	Performance results for sequential compositions . . . . .	173
7.5	Performance results for parallel compositions . . . . .	174
7.6	Performance results for plans with alternating sequential and par- allel execution . . . . .	175
7.7	Effect of composition complexity in performance . . . . .	176
7.8	Effect of adding ramifications to sequential composition plans .	178

7.9	Effect of adding ramifications to parallel composition plans . . .	178
7.10	Effect of adding ramifications to plans with alternating sequential and parallel execution . . . . .	180
7.11	Effect of composition complexity for specifications with ramifica- tions . . . . .	180
7.12	Performance evaluation for functional discovery . . . . .	181
7.13	Performance of pruning for increasingly complex plans . . . . .	183
7.14	Performance of pruning for different success rates and an increas- ing number of implementations per task . . . . .	183
7.15	Performance of pruning for an increasing number of extended plans	185
7.16	Performance of pruning for an increasing number of local goals .	185
7.17	Optimality evaluation for different ranking heuristics . . . . .	186
7.18	Performance evaluation for QoS aggregation (sequential case) . .	188
7.19	Performance evaluation for QoS aggregation (parallel case) . . .	188
7.20	Performance results for the first three phases of WSSL/CVF . . .	193
7.21	Overall performance for increasing numbers of global QoS goals	193

# List of Tables

2.1	Comparison of Automated Service Composition approaches - part 1	58
2.2	Comparison of Automated Service Composition approaches - part 2	59
2.3	Comparison of Automated Service Composition approaches - by category . . . . .	60
3.1	WSSL Specifications for the running example - part 1 . . . . .	87
3.2	WSSL Specifications for the running example - part 2 . . . . .	88
3.3	Surface syntax for WSSL logical expressions . . . . .	90
3.4	XML syntax for WSSL logical expressions . . . . .	95
4.1	Categorization and Aggregation of QoS attributes . . . . .	114
5.1	Mapping from WSSL to FLUX . . . . .	123
5.2	Aggregation of QoS attributes per Execution Path . . . . .	141
6.1	Mapping between WSDL and WSSL . . . . .	156
6.2	Mapping between OWL-S and WSSL . . . . .	163
6.3	Mapping between BPMN and WSSL for composition . . . . .	164
6.4	WSSL/USDL Integration . . . . .	165
7.1	Performance of functional discovery for different cases of overall evaluation . . . . .	191



# Chapter 1

## Introduction

### Contents

---

<b>1.1 Motivation</b> . . . . .	<b>3</b>
1.1.1 Service Specifications . . . . .	3
1.1.2 Representation Problems . . . . .	5
1.1.3 Service Composition . . . . .	6
<b>1.2 Thesis Contribution and Impact</b> . . . . .	<b>7</b>
<b>1.3 Dissertation Outline</b> . . . . .	<b>11</b>

---

The beginning of the 21st century has been marked by two intimately connected paradigm shifts in Computer Science, as regards the way resources become available to end users. The primary paradigm of *Service-Oriented Computing (SOC)* is based on abstracting away from traditional software delivery models and considering software and related data as services available on-demand [Papazoglou et al. 2007]. The service deployment model can be applied to any application component or preexisting code so that they can be transformed into an on-demand, network-available service. The general model of providers licensing applications to customers for a contractually-bound use as a service has been known as *Software as a Service (SaaS)* [Bennett et al. 2000]. A generalization of SaaS to involve any resource, including but not limited to communication, infrastructure and platforms, has led to the advent of *Cloud Computing* [Vaquero et al. 2009], which follows

the principles of distributed computing.

As [Wei and Blake 2010] points out, the two paradigms have a reciprocal relationship: cloud computing provides the computing of services, and SOC provides the services of computing. In both paradigms, services are defined as platform-independent entities that enable access to one or more capabilities, using a prescribed interface and in conformance to constraints and policies [OASIS 2008]; services become available via a certain network, primarily the Internet. Amidst this multitude of available services, the need for a set of design guidelines and principles in SOC endeavors is apparent. This need is satisfied through *Service-Oriented Architecture (SOA)*. SOA is a style of design that guides all aspects of creating and using services throughout their lifecycle. SOA enables an IT infrastructure to allow different applications to exchange data and participate in business processes, regardless of the underlying complexity of applications, such as the exact implementation or the operating systems and programming languages used to develop them.

A SOA can be implemented using several different technologies, the most prominent of which are SOAP [Box et al. 2000], CORBA [Object Management Group 2012a], REST [Fielding and Taylor 2002] and Web services [Booth et al. 2004]. SOAs have been the focus of several standardization efforts by global consortiums such as W3C and OASIS. Web services, in particular, have arguably been the most successful SOA implementation, embraced by software corporations such as IBM and Microsoft which led to them being popular with traditional enterprise.

Worldwide academia has also been a driving force behind service research. Research on services covers a multitude of issues throughout the life-cycle of a service or a Service-Based Application (SBA), typically including service description, discovery, verification, composition, deployment, execution, monitoring and adaptation. This document focuses on research issues related primarily to service description and composition, as well as verification and discovery.



## 1.1 Motivation

### 1.1.1 Service Specifications

Service description deals with specifying all the information needed in order to understand the behavior of a service, as well as access it. Such descriptions should be rich, containing not only functional but also non-functional aspects of the service while they may also contain information on the internal processes of the service, depending on whether the service provider or owner decides to expose such information or not. Service descriptions should also be written in a formal, well-defined language, allowing for automated processing and verification of the produced documents. Service descriptions that capture the aforementioned requirements are referred to as service specifications, to distinguish them from descriptions that are limited to service interfaces.

Formal specifications are indispensable in a variety of service-related activities. Similarly to the case of programming language specifications, service specifications can be used as a basis in order to construct a service. The involved parties (e.g., providers and consumers) can agree on a set of properties about the service and the provider then can implement a service code that satisfies them. On the other hand, if a specification for an implemented service is already available, it can be used to check if it conforms to the specifications agreed upon by the parties involved or if it satisfies a property (such as termination or temporal ordering of actions), realizing so-called service verification [Baryannis et al. 2008]. Conformance and verification checks are an essential part of service auditing processes, which are usually applied to third party or legacy code services in order to determine whether they are suitable for the task at hand.

Moreover, specifications are important when evaluating the results of service adaptation or service evolution. Service adaptation [Benbernou et al. 2008] refers to the process of modifying services in order to satisfy new requirements and to fit new situations dictated by the environment. Service evolution is a more general term that denotes a long-term history of continuous modification of services

after their deployment in order to correct faults, improve performance or other attributes, or to perform adaptation. An example of evaluation for service adaptation and evolution is to ensure that a new version of a service still adheres to the original specification or an evolved specification that has the same or fewer requirements (equal or weaker preconditions) and produces the same or more results (equal or stronger postconditions). In any other case, the adapted or evolved service would be of no use.

Specifications are equally important in the case of composite services, which provide functionality otherwise unattainable by atomic services. Composite services should be made available to consumers in the same way as atomic services, abstracting away complex details of the way participating services are orchestrated to achieve the required functionality. This allows service consumers to invoke services regardless of the way they are implemented (i.e. as an atomic service or as a composition of services). This can be accomplished by providing formal specifications of composite services which present to the end user the minimum information required to understand the functionality offered, often by describing the inputs, outputs, preconditions and effects (collectively known as IOPEs) of the composite service.

Composite specifications also offer great assistance when one attempts to deduce whether a set of services can actually be composed in a meaningful way. During the process of creating composite specifications, inconsistencies may be detected between preconditions and/or postconditions of the participating services, rendering that particular set of services not composable. Thus, such problems can be prevented before the composite service is delivered to the end user, so that they may be resolved by replacing the service or services that cause the inconsistencies.

The need for service specifications instead of simple service descriptions has also been recognized in service literature, most notably and recently in [Terlouw and Albani 2013] where the authors define a generic framework for service specifications based on modeling organizations as social systems and services as activ-

ities that support the actors of these systems. While [Terlouw and Albani 2013] deals primarily with deciding *what* should be specified for a service, the present thesis offers a solution to the problem of *how* to specify a service, taking into account a series of important representation problems that are defined in the sequel.

### 1.1.2 Representation Problems

Service specifications rely on the expression of conditions that should hold before and after service execution. Such specifications are prone to a family of problems, which are known in the AI literature as the frame problem, the ramification problem and the qualification problem. The frame problem concerns itself with expressing the non-effects of an action, i.e. what remains unchanged afterwards. The ramification problem deals with the modeling of knock-on and indirect effects, collectively known as ramifications. Finally, the qualification problem, in contrast to the other ones, deals with the representation of preconditions and the inability to take into account any possible condition under any circumstance.

While research in other fields such as programming specifications or reasoning about action and change has come up with satisfying solutions to these problems, research in services has largely ignored them, at the same time ignoring their effects. Specifically, failing to address the frame problem results in service descriptions that cannot guarantee service consumers that the specified effects are the only ones produced after a service execution. This essentially compromises service reusability in composite applications, while also raising several important issues such as integrity of the provided service, provider trustworthiness, as well as safety and privacy of the users and data involved.

Moreover, disregarding ramifications of a service results in an incomplete service behavior model where side effects are not taken into account, resulting in similar pitfalls as the ones caused by the frame problem. Especially in the case of service composition, ignoring the ramification problem can lead to incorrect

composite services, since a ramification of one service can violate a precondition of a subsequent service in the composite process.

Moving on from consequences related to the effects of a service execution, the qualification problem introduces issues with regard to the ability to explain service executions that are abnormal and unexpected. Without a solution, there is no way to justify that a service has not produced the expected results, even though all preconditions included in the specification are satisfied at execution time. All the aforementioned issues caused by the frame, ramification and qualification problems substantiate the imperative need for a service specification language that offers ways to address them.

### **1.1.3 Service Composition**

Service composition involves combining and coordinating a set of services with the purpose of achieving added-value functionality that cannot be realized through existing services. The process of creating a composition schema in order to satisfy a series of goals set by a requester is a really complex and multifaceted problem, since one has to deal with many different issues, such as searching in ever-growing service repositories, resolving any conflicts and inconsistencies between chosen services and adapting to the dynamic characteristics of service-based systems, with services going offline, new services coming online, and existing services changing their characteristics.

Attempting to overcome all these problems manually leads most certainly to a composition schema that is not fault-proof while the whole procedure consumes a lot of time and resources and is effectively rendered unscalable. This had led to the consensus that a certain degree of automation is necessary in the composition procedure. Automated service composition has attracted a great deal of research effort worldwide. In spite of this, several research challenges are yet to be addressed by any of the proposed service composition approaches in literature.

First and foremost, the frame, ramification and qualification problems affect

service composition as well, as it was analyzed earlier, since the composition process relies on complete behavioral specifications of services in order to be able to decide whether services can be composed without resulting in conflicts and correctness violation, as well as whether the resulting composition satisfies a given goal. Hence, a service specification language that addresses these representation problems is also of great use and importance to service composition approaches.

Furthermore, it is common for service composition research approaches to focus on a specific subset of requirements that is pertinent to the research line followed and satisfy these, while at the same time ignoring any other requirements of equal importance and the correlation among them. For instance, there are composition approaches that support multiple composition patterns including complex ones such as conditional and iterative execution, but take into account only functional goals, resulting in composite services that may not satisfy quality constraints. Thus, it is challenging to realize a service composition process that is able to satisfy a maximal set of those requirements that have been deemed research-worthy in literature.

## 1.2 Thesis Contribution and Impact

To address the aforementioned issues in service description, we propose a novel language for the specification and composition of services, named the Web Service Specification Language (WSSL, pronounced /'wi:zəl/). The language is designed with the explicit purpose of describing service behavior by means of complete specifications that take into account the representation problems mentioned previously. WSSL's foundation is the fluent calculus [Thielscher 2005b], a specification language and system for robots that offers solutions to these problems. More specifically, WSSL is designed so that it:

- relies on a description model that is independent of service design models,
- provides solutions to the frame, ramification and qualification problems

that are directly connected to the foundations of the language,

- realizes semantic awareness, linking specification elements to ontology concepts through IRIs,
- supports the specification of service compositions, taking into account fundamental workflow patterns, including non-deterministic ones such as conditionals and loops,
- realizes service composition through planning in the fluent calculus with FLUX,
- includes specification of QoS aspects,
- supports partial observability of states as well as the more generalized notion of knowledge states,
- offers grounding and translation mechanisms to existing service description languages (WSDL, OWL-S, WSMO) as well as USDL integration,
- facilitates specification document parsing and exchange via the definition of an XML-based syntax.

To demonstrate the power of WSSL and the significant benefits it provides, we also propose an integrated composition and verification framework, WSSL/CVF, that relies on WSSL specifications. The proposed framework possesses the following innovative characteristics:

- The composition process takes advantage of complete behavioral specifications of services expressed in WSSL, allowing for services and goals that include ramifications.
- The verification process also benefits from the solutions to the frame, ramification and qualification problems in two ways:
  - Behavioral properties that refer to preconditions, effects and ramifications, and not only to inputs and outputs, can be verified.

- Verification also takes into account abnormal and unexpected cases, apart from the normal service behaviour, providing the ability to explain such cases, rather than consider them incompatible with the specification.
- A novel functional discovery approach for services specified using WSSL is included with the following distinctive features:
  - Due to WSSL's solutions to the frame, ramification and qualification problems, discovery relies on complete service behaviour models that include, for instance, ramifications and non-effects, safeguarding from inconsistent or inaccurate discovery solutions.
  - Due to the unified nature of WSSL specifications, taking into account all aspects of service behaviour during discovery does not require a different approach for each aspect, resulting in improved performance.
  - Discovery is applied on specification repositories, grouping implementations of the same functionality under a single behaviour specification, further boosting performance.
- The WSSL foundations of the framework allow for the simultaneous satisfaction of several important requirements, namely automation, dynamicity, correctness and support for semantics, non-determinism and partial observability.
- QoS-awareness is realized through the definition of pruning and ranking techniques that combine heuristics and task-specific QoS goals and are applied to plans that result from functional composition and discovery.
- An all-encompassing QoS aggregation approach is included, based on a classification of QoS attributes depending on their value types and nature that is derived from the analysis of QoS aspects in [Kritikos and Plexousakis 2009b].

An extensive experimental evaluation of the framework is performed, in order to investigate performance scalability in terms of execution time and memory consumption, as well as optimality with regard to the produced composite process. Evaluation relies on synthetically generated WSSL specifications and composition problems, in order to cover a wide spectrum of specification complexity. Results show that good performance is guaranteed even for problems that exceed real-world requirements.

WSSL and the accompanying composition and verification framework can have a substantial impact to SOA stakeholders. With WSSL, service providers are able to provide complete specifications of what they are offering, which enables them to more effectively advertise their service products to potential clients. In this way, service providers are more likely to be trusted, since they offer more details about what they provide instead of a simple input/output service interface. Of course, the possibility of providing false specifications cannot be ruled out.

Service consumers, on the other hand, are informed of the exact way in which a service is expected to perform, as well as the produced results, which enables them to make knowledgeable choices and select the service that is the most suitable match for their requirements. In safety-critical systems, this is of utmost importance since choosing a service that fails to achieve the complete set of specification requirements may potentially have serious repercussions. If a suitable service does not exist, then a new one can possibly be composed according to the requirements. The nature of WSSL specifications facilitates automation, speeding up the process and at the same time lowering costs, making SBAs more attractive to industry stakeholders.

The proposed composition and verification framework can be of great assistance to stakeholders in service engineering, especially SBA designers and composition architects. At the cost of an increased effort in creating service specifications, the composition design and modeling effort can be significantly reduced, at the same time mitigating the effects of human error. Nonetheless, the framework relies on input from designers in defining suitable heuristics, for both functional



composition and plan pruning processes. Hence, it does not replace the service designer but rather automates the process of obtaining optimal service compositions for a suitably defined composition problem.

### 1.3 Dissertation Outline

The rest of this document is organized as follows. Chapter 2 provides an indicative scenario that is used as a running example throughout the dissertation, followed by an extensive analysis of the necessary background knowledge. This includes a definition of the frame, ramification and qualification problems and an overview of solutions that have been proposed in literature, an overview of the most outstanding service description efforts, a definition of requirements for service composition and a comparative analysis of the state-of-the-art in automated service composition, as well as the most prominent works in service discovery and verification.

In Chapter 3 the main contribution of this thesis, the Web Service Specification Language, is presented. First, the fluent calculus foundations of the language are briefly described, followed by a formal definition of the abstract syntax and semantics. Afterwards, a surface syntax, as well as an XML syntax are provided. Chapter 4 introduces a series of extensions to the initial language definition. The extensions cover three distinct directions: specification of composite services and support for service composition; specification of QoS profiles; and handling partially observable states.

Chapter 5 proposes an implementation of WSSL using the logic programming language FLUX which acts as the basis for the definition and analysis of a complete QoS-aware service composition and verification framework, named WSSL/CVF. Chapter 6 performs an examination of a series of properties of WSSL and the accompanying framework, focusing on correctness, decidability and complexity, as well as language applicability in terms of WSSL's connection to other efforts in service science. A thorough evaluation of WSSL/CVF as a whole, as well as the

individual components it comprises, is conducted and presented in Chapter 7. Finally, Chapter 8 summarizes the main contributions of this thesis and points out future research directions.

Research related to this thesis has resulted in the following publications:

- An initial version of the background analysis in service description and the state-of-the-art in automated service composition in Chapter 2 appears in [Baryannis and Plexousakis 2010a] and [Baryannis et al. 2010].
- A preliminary analysis of some of the research challenges that are addressed in this thesis appears in [Baryannis and Plexousakis 2010b].
- The theoretical foundations of WSSL's extension to support composite patterns in Section 4.1 is based on the method for deriving composite service specifications that is presented in [Baryannis et al. 2012].
- The initial definition of WSSL in terms of its syntax and semantics in Chapter 3 and parts of the implementation in Section 5.1 appear in [Baryannis and Plexousakis 2013].
- The extension of WSSL to support composition in Section 4.1 and the implementation of a WSSL-based composition planner in Section 5.1 have been published in [Baryannis and Plexousakis 2014].
- The extensions for quality and uncertainty in Chapter 4 as well as the implementation and evaluation of the proposed composition and verification framework in Section 5.2 and Chapter 7 are included in [Baryannis et al. 2014], under review at the time of writing the dissertation.

## Chapter 2

# Background and Literature

## Review

### Contents

---

<b>2.1</b>	<b>Running Example . . . . .</b>	<b>15</b>
<b>2.2</b>	<b>The Frame, Ramification and Qualification Problems . . .</b>	<b>18</b>
2.2.1	Definitions . . . . .	18
2.2.2	The Temporal Action Logic case . . . . .	21
2.2.3	Modular-E: The Event Calculus case . . . . .	22
2.2.4	The Fluent Calculus case . . . . .	23
2.2.5	Comparison . . . . .	24
<b>2.3</b>	<b>Service Description and Specification . . . . .</b>	<b>25</b>
2.3.1	Web Services Description Language (WSDL) . . . . .	25
2.3.2	Semantic Annotations for WSDL (SAWSDL) . . . . .	27
2.3.3	OWL-S: Semantic Markup for Web Services . . . . .	28
2.3.4	Web Service Modeling Ontology (WSMO) . . . . .	30
2.3.5	Semantic Web Services Framework (SWSF) . . . . .	32
2.3.6	The SAVVY-WS Framework . . . . .	33
2.3.7	Unified Service Description Language (USDL) . . . . .	34
2.3.8	Conclusion . . . . .	35

---

<b>2.4 Automated Service Composition</b>	<b>36</b>
2.4.1 Requirements	37
2.4.2 Workflow-based Approaches	43
2.4.3 Model-based Approaches	46
2.4.4 Logic-based Approaches	47
2.4.5 AI Planning Approaches	49
2.4.6 Comparison and Research Challenges	56
<b>2.5 Specification-based Service Discovery</b>	<b>61</b>
2.5.1 Discussion	64
<b>2.6 Verification of Service Behavior</b>	<b>65</b>
2.6.1 Discussion	67

---

This chapter provides the necessary background knowledge as well as an extended literature review pertaining to the topics examined in this thesis. First, a service composition scenario is presented with the purpose of using it as a running example throughout the dissertation. Then, the main objective of the thesis is analyzed, through the definition of the frame, ramification and qualification problems, as well as the various solutions that have been proposed in literature. An overview of the most outstanding service description efforts follows. Afterwards, a set of requirements for automated service composition is defined and used as a basis for the comparative analysis of the state-of-the-art in service composition that follows. The comparison yields a series of research challenges that drive the rest of this thesis. Finally, a concise review of research related to specification-based service discovery and verification of service behavior is offered, since both topics are highly relevant to the language and framework presented in this dissertation.

## 2.1 Running Example

In this section, we present an indicative scenario that illustrates the motivation behind this thesis in general and more specifically the need for an expressive specification language, such as WSSL, to realize and facilitate service description, composition and verification. The scenario is inspired by HelpMeOut, a process for vehicle drivers to get assistance in case of an emergency, presented in [Ali et al. 2011] and involves designing and verifying a composite process that covers all necessary steps relative to vehicle road assistance. In our scenario, vehicle drivers in need of assistance communicate with a call center, which collects all pertinent information in order to find and dispatch the most suitable repair center employee. Payment and reporting steps follow after the issue has been resolved.

We assume that we have access to a service specification repository containing multiple service specifications that realize the individual required functionalities, each one with varying QoS features. The composite service must satisfy a series of requirements, involving both functional and non-functional aspects:

- Requests for assistance can be received either through a call center or through SMS but each process is associated with only one of these methods of communication. Information about the driver location as well as the vehicle status is processed in combination in order to drive a separate procedure that determines the nearest mechanic able to assist in the current situation.
- The payment process that follows the on-site actions should support credit card payment as well as trigger card deactivation in case a daily spending limit has been reached.
- An incident report must be generated and delivered to the driver, through electronic or traditional mail, depending on the driver's choice.
- The results of the executed composite process should be specified and explained, even under incomplete information (e.g., the driver's choice for report delivery is not known beforehand) or unforeseen circumstances (e.g.,

no report is delivered by e-mail, as per the driver's wishes, even though all tasks in the process have completed successfully)

Apart from functionality, the resulting composite service should also achieve a QoS level defined by the following requirements:

- Availability of the services implementing the tasks of receiving calls and SMS for assistance should be at least 0.99.
- The service implementing the search for a suitable mechanic should complete in less than 60 seconds.
- The payment process must support integrity and security features.
- The overall cost for the composite process should not exceed 10€, while overall throughput must be at least 50 requests/min.

Given a service specification repository containing specifications for services that implement the separate tasks described above, as well as a specification of the initial state, we would like to automatically create a composite process that satisfies all the aforementioned requirements and realizes the complete HelpMe-Out road assistance scenario. An example process is shown in Fig. 2.1.

Satisfying all requirements depends upon a service composition framework that supports rich service descriptions, in the form of specifications that take into account the frame, ramification and qualification problems, so that non-effects, ramifications and explanations for unexpected behavior of services can be expressed and considered during composition. The discussion in the rest of this chapter centers around the claim that current service description and composition efforts are not adequate enough to realize the goals of the road assistance scenario we defined.

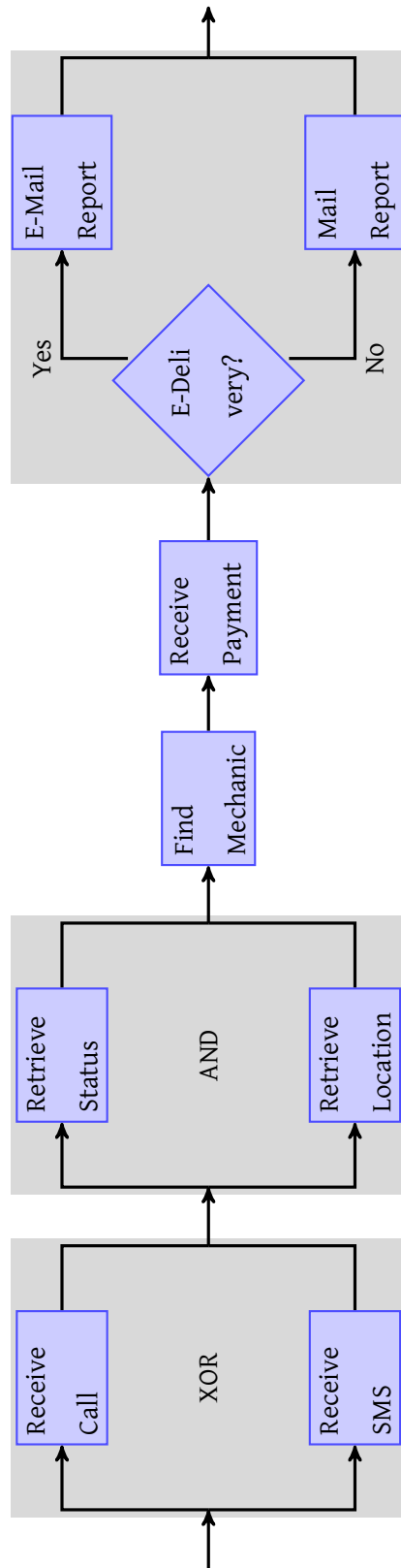


Figure 2.1: Composite process of the running example

## 2.2 The Frame, Ramification and Qualification Problems

As already mentioned in the introductory chapter, service specifications are affected by a family of problems that includes the frame, ramification and qualification problems. In this section, we define all three problems and offer a comparative analysis of the solutions that have been proposed in literature. We focus only on solution schemas that target all three problems rather than a subset of them. Based on this analysis, we plant the seeds for the selection of the fluent calculus as a basis for the service specification language that we propose in Chapter 3.

### 2.2.1 Definitions

#### The Frame Problem

As it was first identified in [Borgida et al. 1995], formal specifications that employ the precondition/postcondition notation are prone to the frame problem [McCarthy and Hayes 1969], a long-standing problem in the field of Artificial Intelligence. The frame problem stems from the fact that including clauses that state only what is changed when preparing formal specifications is inadequate since it may lead to inconsistencies and compromise the capacity of formally proving certain properties of specifications. One should include additional clauses, called frame axioms, explicitly stating that apart from the changes declared in the rest of the specification, nothing else changes. Solving the frame problem means expressing frame axioms without resulting in extremely lengthy, complex, possibly inconsistent, obscure specifications and at the same time retaining the ability of proving formal properties of the specifications.

To illustrate the effects of the frame problem, consider the payment process of the running example. According to the requirements, credit card payment should be supported. Hence, any service execution that implements the payment process should be associated with a credit card and withdraw money from the account associated with that credit card. Due to the sensitive nature of this procedure, we



need not only a guarantee that the service executes the payment process successfully but also that no other accounts are affected by it. In other words, we need a specification language that guarantees that the specified effects are the only ones that result after a successful execution.

### **The Ramification Problem**

The frame problem is closely related to the ramification problem, defined in [Miller 2006] as the problem of adequately representing and inferring information about the knock-on and indirect effects (also known as ramifications) that might accompany the direct effects of an action or event. The relation of the frame and ramification problems is somewhat contradicting: if one deals with the frame problem by expressing that no other effects are allowed except the ones explicitly stated, then any indirect effects are disallowed by definition, hence any solution to the ramification problem is precluded.

Returning to the running example and the associated payment process, recall that one of the stated requirements is that credit card deactivation must be triggered in case a daily spending limit has been reached. Thus, we need a specification language that can express not only effects but also side-effects that result from them. Note that if the solution to the frame problem precludes any other effects other than the direct ones, then any solution to the ramification problem is, by definition, impossible. It is apparent that any solution to the ramification problem in service specifications should be examined in correlation to the frame problem.

Ramifications must be included in a service specification since a potential service consumer needs to be aware of all (direct and indirect) effects of executing the service. This is particularly important in service compositions, as the lack of knowledge of an indirect effect may lead to the assumption that a composition is valid and correct while that particular effect may contradict a precondition of another participating service, leading to an inconsistent composite service. For instance, in the running example, if we are unaware of the ramifications of the

payment service, then we can assume that, under any circumstance, we can execute the payment service consecutively (possibly for different composite services) without violating correctness. This is not the case, however, if the first execution leads to the credit card being deactivated because the daily limit has been reached.

### **The Qualification Problem**

While the ramification problem deals with the effects of an action, the qualification problem deals with circumstances and conditions that must be met prior to the execution of an action. In [Miller 2006], two major aspects are defined separately, the endogenous and the exogenous qualification problem. The endogenous qualification problem is the problem (and sometimes the impossibility) of listing all preconditions (also known as qualifications) that must be satisfied for an action to have the desired effect and at the same time updating these qualifications as new statements become part of our knowledge, without resulting in inconsistencies. On the other hand, the exogenous qualification problem deals with qualifications that are outside the scope of our theory and result in contradicting some effects of an action due to inconsistencies.

In the context of services, it can be argued that even if one considers a service specification as complete, there might always be some conditions that have not been explicitly declared. Trying to declare more and more preconditions may then lead to a service that is not executable, or to specifications that are rendered inconsistent once a new statement becomes part of our knowledge. As far as the exogenous aspect is concerned, service execution can be affected by numerous external factors (as opposed to "internal" ones, i.e. related to service functionality), such as failures at the underlying infrastructure or authentication issues in data access, to name but two. In this thesis, we deal with the exogenous qualification problem only, since the endogenous aspect is closer to the philosophical aspect of qualifications rather than the practical one that is more related to the service world.

Going back to the running example, suppose that in an execution of the road assistance composite service, we have established that all preconditions for the report delivery sub-process hold; we are also aware that the driver has chosen to receive the report via e-mail. Thus, we expect that, at the end of the process, a report is delivered electronically to the user. If, however, due to some unforeseen circumstance, we find out that no report has been delivered, we result in an inconsistency between the service specification and the actual observed behavior. To address this issue, service specifications must be able to account for qualifications that are outside the scope of normal service execution and represent deviations to the normal case, in order to describe service behavior in a consistent and complete manner.

### 2.2.2 The Temporal Action Logic case

In [Doherty 1994], [Gustafsson and Doherty 1996] and [Kvarnström and Doherty 2000], the authors propose extensions to Temporal Action Logic (TAL) to deal with the frame, ramification and qualification problems, respectively. TAL and its variants use the notion of preferential entailment for reasoning about action and change, which essentially applies a strict partial order to logical interpretations to decide which models to keep. TAL uses narrative scenarios based on states and has an explicit notion of time.

The solution to the frame problem is based on the notion of occlusion: whether a feature may change value or not is defined by special Occlude predicates which explicitly express the so-called “permission to change value”. The Occlude predicate also provides the insight for a solution to the ramification problem, since the authors realize that it could also be used to express causal rules between effects. These specialized fluent dependency constraints led to an extension called TAL-RC which successfully expresses ramifications that refer to any part of the domain that is modeled, for both boolean and non-boolean fluents.

A further extension, called TAL-Q, was designed in order to deal with the qual-

ification problem. For each action in the theory, a strong qualification is included, which represents all qualifications that are true in normal situations, and can only be made explicitly false if an associated dependency constraint becomes valid. They also allow for the expression of weak qualifications, defined by the authors as conditions that could lead to an action but only if they are considered in correlation with other domain and dependency constraints. Thus, strong qualifications are necessary for an action to be successful, while weak ones need to be examined in correlation with other constraints.

### 2.2.3 Modular-E: The Event Calculus case

In [Kakas et al. 2011], the authors present  $\mathcal{M}odular\text{-}\mathcal{E}(\mathcal{M}\mathcal{E})$ , a specialized logic closely related to the event calculus, for reasoning about actions with the explicit requirement that it addresses both the endogenous and exogenous aspects of the qualification problem. Due to the inextricable link among all three problems,  $\mathcal{M}odular\text{-}\mathcal{E}$  offers a robust and complete solution to the frame and ramification problems too, based on the notion of elaboration tolerance as expressed in [McCarthy 1999] and its particular characteristic that is labeled as “free will”.

$\mathcal{M}\mathcal{E}$  deals with ramifications by considering chains of instantaneous, temporary transition states, implied by causal laws and propositions of the form  $C$  causes  $F$ , where  $C$  and  $F$  are fluents. Within these causal chains, processes are introduced to describe initiation and termination of fluents. This allows to treat all possible micro-orderings of effects, as well as competing and looping effects. Inconsistencies are allowed during such processes, but they must be resolved until the end of the process. In addition,  $\mathcal{M}\mathcal{E}$  carries over the event calculus notion of default persistence, allowing change only when action laws or ramifications justify it, thus offering a solution to the frame problem.

As far as the qualification problem is concerned, the endogenous aspect is realized by the modular and elaboration tolerant features of  $\mathcal{M}\mathcal{E}$ . Propositions of the form *always*  $A$  allow for the expression of global qualifications, without

having to explicitly add them as local qualifications to the actions they affect. All global and local qualifications are assimilated in the theory in order to assess whether an action is successful or not. Note that due to  $\mathcal{ME}$ 's notion of "free will", an action occurrence is synonymous to an action attempt: any action can be attempted at any time, without any guarantee of success.

Finally, the exogenous qualification problem is dealt with by extending the theory even further with propositions of the form *normally*  $N$  with the semantics that the fluent  $N$  is true under normal circumstances. Exogenous qualifications for an action are expressed using a fluent of the form  $NormExo(Law_{Id})$ , where  $Law_{Id}$  is the identifier of the *causes* proposition related to the action. To express that all exogenous qualifications are assumed to be true by default, one simply states that *normally*  $NormExo(Law_{Id})$ .

#### 2.2.4 The Fluent Calculus case

Research in the fluent calculus ([Thielscher 1997; 1999; 2001b], presented in a complete form in [Thielscher 2005b]) attempted to first solve the ramification problem, followed up with solutions to the frame problem and the qualification problem. All solutions involve augmenting the initial definition of the fluent calculus with constructs specifically designed to address the issues caused by each problem, namely state update axioms for the frame problem, causal propagation of indirect effects for the ramification problem and abnormal qualifications as well as default reasoning for the qualification problem.

Thielscher defines causal relationships in the fluent calculus as expressions of the form  $(\forall)(\Gamma \rightarrow Causes(z, p, n, z', p', n', s))$  where  $z, p, n, z', p'$  and  $n'$  are state variables and  $\Gamma$  is a first-order formula. The semantics is that, under conditions expressed by  $\Gamma$ , the positive and negative effects  $p$  and  $n$  that have occurred cause an automatic update from state  $z$  to state  $z'$ , with positive and negative effects  $p'$  and  $n'$ . Such a production of effects can be done repeatedly, producing new effects from already acquired indirect effects.

Thielscher then focused on improving an existing solution to the frame problem in the fluent calculus that is equivalent in its expressive power to the solution of successor state axioms in the situation calculus [Reiter 1991] but also solves the inferential facet of the frame problem as well, providing a robust, systematic way of producing these axioms. State update axioms, as they are named, are of the general form  $\Delta(s) \supset State(Do(A, s)) \circ \theta^- = State(s) \circ \theta^+$  and have the semantics that the execution of action A at state s under condition  $\Delta(s)$  results in only the specified positive ( $\theta^+$ ) and negative ( $\theta^-$ ) changes (a positive change makes a fluent true, while a negative one makes it false). Provided that the effects are finite, the author provides a systematic procedure of deriving state update axioms from given effect axioms in the situation calculus.

Finally, the qualification problem is dealt with by first explaining how the straightforward solution of simply assuming away all abnormal qualifications by default may lead to anomalous models when an action itself causes abnormal circumstances. The proposed solution is to treat the qualification problem while respecting causality: abnormal qualifications are assumed not to hold initially and not to arise in later situations *unless they are caused*. In order to realize that, a solution to the frame and ramification problems is a prerequisite, hence the fluent calculus is a suitable candidate. Special fluents for expressing abnormal qualifications are employed, explicitly indicating the cause as a specific action or an exogenous factor. Default theory is employed to express that, by default, abnormal qualifications do not hold in the initial state and exogenous ones do not hold in any state.

### 2.2.5 Comparison

$\mathcal{ME}$  is the most recent work that claims to address all three problems and, as such, is advantageous to the bodies of work in the fluent calculus and Temporal Action Logic, since the authors specifically addressed the weaknesses of previous work. In comparison to the fluent calculus approach,  $\mathcal{ME}$  has the advantage of

covering the case of failed actions or action attempts due to the notion of “free will”. Finally, although a comparison between  $\mathcal{ME}$  and TAL-Q is deemed difficult by the authors, they nevertheless point the fact that TAL-Q is only able to handle off-line planning and prediction problems and at the same time does not handle the exogenous qualification problem.

However, with respect to the domain of Web services (and services in general), the solutions offered in the fluent and event calculi are somewhat equivalent, since the advantage of expressing failed actions or action attempts is not vital: in the service world we only need to express the results of a successful execution, the conditions under which they are achieved and explanations for any unforeseen execution based on abnormal qualifications. Based on this, the fluent calculus seems a more suitable candidate for the purposes of this thesis, as we justify in more detail at the beginning of Chapter 3.

## 2.3 Service Description and Specification

We continue the analysis of related literature with a concise overview of the most important efforts to handle the issue of describing and specifying services and service compositions. These include WSDL, SAWSDL, USDL, OWL-S, WSMO, as well as the more recent SAVVY-WS framework. The focus of this presentation is to evaluate if and to what extent these efforts address the problems analyzed in the previous section.

### 2.3.1 Web Services Description Language (WSDL)

Web Services Description Language (WSDL) [Chinnici et al. 2007] is a W3C Recommendation, now in its second version, that has been established as the de facto description language for Web services. The four fundamental elements of WSDL are the following:

- *types*: describes the kinds of messages that the service sends and receives.
- *interface*: offers an abstract description of the provided functionality.
- *binding*: describes **how** to access the service.
- *service*: describes **where** to access the service.

WSDL interfaces offer an abstract way of describing the service functionality, in contrast to the concrete representation offered by the binding and service elements. A WSDL interface consists of a set of operations, each one of them describing a simple interaction between the service and the client, producing a set of outputs given a set of inputs. The description is realized through a message exchange pattern that is followed when the particular operation is invoked. WSDL 2.0 contains eight predefined message patterns but new ones can easily be defined.

A WSDL binding specifies concrete message format and transmission protocol details for an interface. Each operation in a WSDL description must be associated with a binding. Several typical binding extensions are defined in WSDL 2.0, such as SOAP binding or HTML binding; the service provider is free to use others, provided that they are supported by potential clients and service toolkits. Finally, a WSDL service element specifies a single interface as well as an associated binding that the service supports, along with one or more endpoint locations where that service can be accessed.

From this brief description, it should be obvious that, even in its second version, WSDL remains solely a language for the syntactic description of Web services. For WSDL, a service is merely a construct that produces a specific set of outputs, given a specific set of inputs. Without doubt, this does not allow service consumers to understand what a service does and under which conditions, making it difficult to decide whether a particular service is suitable for their purposes or not. Moreover, the lack of semantic capabilities and composite service support renders WSDL incapable of assisting in any automated service discovery and com-



position framework. The fact that WSDL does not even consider preconditions and postconditions renders the discussion on the three problems described in the previous chapter inapplicable.

### 2.3.2 Semantic Annotations for WSDL (SAWSDL)

SAWSDL [Farrell and Lausen 2007] is based on and shares the same design principles of previous work published as a W3C Member Submission with the title Web Service Semantics (WSDL-S) [Akkiraju et al. 2005], headlined by IBM and the LSDIS Lab of the University of Georgia. SAWSDL defines a way to semantically annotate WSDL interfaces and operations as well as XML Schema types, linking them to concepts in an ontology or a mapping document. The annotation mechanism is independent of ontology or mapping languages and can be applied to both WSDL 1.1 and WSDL 2.0 documents, although the latter case is more seamless than the former.

SAWSDL offers two basic semantic annotation constructs, through the extension attributes `modelReference` and `schemaMapping`. The *modelReference* attribute allows multiple annotations to be associated with a WSDL or XML Schema component via a set of URIs. These URIs may identify concepts expressed in different semantic representation languages. `modelReference` attributes can be used in WSDL interfaces and operations, or XML Schema elements, types and attributes.

As far as the second annotation mechanism is concerned, two attributes are provided: *liftingSchemaMapping* and *loweringSchemaMapping*. These are used to address post-discovery issues, since mismatches may still exist between the semantic model and the structure of inputs and outputs. Schema mapping relates the instance data defined by an XML Schema document with some semantic data defined by a semantic model (e.g., an ontology). While *liftingSchemaMapping* defines how an XML instance document is transformed to data that conforms to some semantic model, *loweringSchemaMapping* follows the opposite direction, defining how data in a semantic model is transformed to XML instance data.

SAWSDL attempts to provide semantic capabilities to existing WSDL descriptions, electing to keep the semantic model outside WSDL, making the approach independent from any ontology language. However, without describing how the use of annotations in different languages relate to one another, it is rather difficult, if not impossible to formally define requests, queries or matching between service requests and service descriptions. As a result, SAWSDL may be successful in annotating WSDL with semantic information, but does not offer any support for automated service discovery and composition. Moreover, due to the fact that it uses the description model of WSDL as-is, SAWSDL inherits the limitations of input-output descriptions and the fact that it cannot be employed to address the issues raised in this thesis.

### 2.3.3 OWL-S: Semantic Markup for Web Services

OWL-S [Martin et al. 2004] was the first attempt to establish a framework within which Web service descriptions are created and shared, employing a standard ontology, consisting of a set of basic classes and properties for declaring and describing services. The ontology structuring mechanisms of OWL provided an appropriate, Web-compatible representation language to create the standard ontology of OWL-S.

In Fig. 2.2, the structuring of OWL-S in sub-ontologies is shown. Service Profile allows service providers to advertise the services they offer, in such a way that service requesters can easily find what they are looking for. Service providers are free to include whatever information they deem necessary, but the OWL-S standard offers a predefined subclass that should contain information on the service provider, the service functionality and a set of service characteristics. The functional description of a Web service in OWL-S contains the full set of IOPEs, in contrast to WSDL.

The Service Model sub-ontology in OWL-S describes service functionality and specifies the manner in which a client may interact with the service in order to

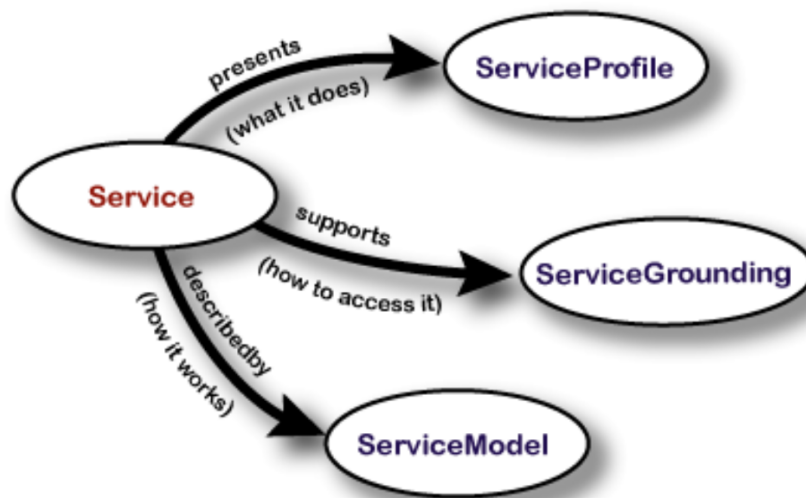


Figure 2.2: Top level of the OWL-S ontology [Martin et al. 2004]

consume its functionality. It is usually a superset of the information contained in a Service Profile, since a profile contains only the information that is necessary in order to advertise the Web service. Again in contrast to WSDL, OWL-S provides a means to specify a Web service as a business process or a workflow. This process can represent a single Web service (called atomic process) or a composite one. A composite process is specified using control constructs such as Sequence, Split-Join or If-Then-Else, representing sequential, parallel and conditional composition schemas respectively.

OWL-S provides a backward compatibility mechanism through the Service Grounding sub-ontology, which allows service designers to map an OWL-S description to existing service description languages, such as the most prominent one, WSDL. This mechanism is intended to allow existing WSDL descriptions to be reused and augmented using OWL-S, at the same time delegating actual service access to the existing mechanisms offered by WSDL.

OWL-S made strides in the correct direction regarding service specifications, since it was the first service description framework to employ IOPEs, offer semantic capabilities and support composite service description. Thus, it is the first effort that is directly related to the issues raised in the previous section but, unfor-

tunately, it fails to offer effective solutions to the frame, ramification and qualification problems. While it supports preconditions and postconditions, it does not provide any way to specify non-effects or indirect effects, while there is no mention of explaining abnormal executions.

### 2.3.4 Web Service Modeling Ontology (WSMO)

WSMO [Roman et al. 2005] is a conceptual model for describing various aspects related to services in the Semantic Web, initially produced by the ESSI WSMO working group, with further research conducted by the European project SOA4All. The objective of WSMO and its accompanying efforts is to solve the application integration problem for Web services by defining a coherent technology for Semantic Web services. As illustrated in Fig. 2.3, four main components are defined in WSMO and are outlined in the rest of this section.

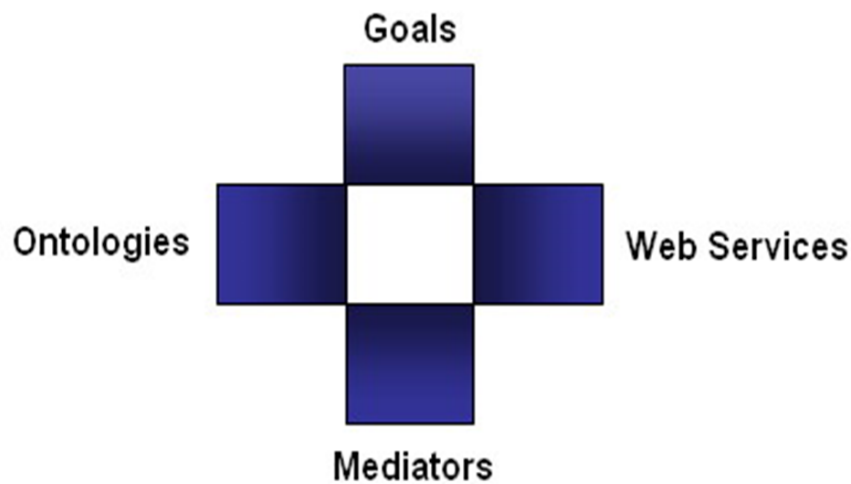


Figure 2.3: The four main WSMO components [Roman et al. 2005]

WSMO Ontologies provide domain specific terminologies for describing the other elements. The basic blocks of an ontology are concepts, relations, functions, instances, and axioms. Web services description in WSMO is realized using two different viewpoints: interface and capabilities. WSMO capabilities are similar to the

IOPE model used in OWL-S. A capability should contain preconditions and assumptions that must be true before the execution of the service, postconditions that are true if the service has executed successfully, and effects that illustrate how service execution changes the world state. WSMO Mediators connect heterogeneous components of a WSMO description when one encounters structural, semantic or conceptual incompatibilities. Mismatches can arise at the data or process level.

A WSMO interface describes how service functionality can be achieved. Two complementary views of the operational competence of a Web service are provided, namely orchestration and choreography. Orchestration offers a description of how the overall functionality is achieved by means of cooperation of different Web service providers, while choreography is essentially a description of the communication pattern that allows one to consume the service functionality.

WSMO Goals define exactly what a service requester demands from a Web service to offer. The requester (whether it is a human or an agent) defines a set of goals and the corresponding system tries to find services or combinations of services that can realize all of them. In order to produce a complete goal description, one needs to describe both the interface and the capabilities of a Web service that, if it existed, it would completely satisfy the request at hand. In this way, requests and services are strongly decoupled, since requests are based directly on goal descriptions.

WSMO presents an alternative way to model Semantic Web services and support automated service composition. Like OWL-S, it employs IOPEs and supports composition description. Unlike OWL-S, it supports both orchestration and choreography views of a composite service. The WSMO working group also goes one step further and acknowledges the existence of the frame problem in WSMO specifications. However, the proposed solution essentially avoids the frame problem by applying restrictive update formalisms. Also, in a document prepared by the WSMO working group [Keller and Lausen 2006], the authors propose to label postconditions as incomplete, when no action has been taken to address the frame problem; again, this is a circumvention of the frame problem, rather than a solu-

tion to it. Moreover, the ramification and qualification problems are not considered at all.

### 2.3.5 Semantic Web Services Framework (SWSF)

SWSF [Battle et al. 2005] is another effort to realize the Semantic Web services vision and is influenced by both OWL-S and WSMO. SWSF comprises two major components, the Semantic Web Services Ontology (SWSO) and the Semantic Web Services Language (SWSL). SWSL is used to specify formal characterizations of Web service concepts and descriptions of individual services. It includes two sub-languages, the first of which is named SWSL-Rules, is based on logic-programming and rule-based reasoning and is used to support the use of service ontologies in reasoning and execution environments, providing a variety of tasks that range from service profile specification to service discovery, contracting, policy specification and so on. The second sub-language of SWSL, SWSL-FOL, is based on first-order logic and is used primarily to express the formal characterization of Web service concepts.

SWSO presents a conceptual model by which Web services can be described and which, for the most part, is really similar to OWL-S. It is divided into three major components: Service Descriptors, Process Model and Grounding. Service Descriptors are the equivalent to OWL-S Service Profile while Grounding is equivalent to OWL-S Service Grounding.

The SWSO Process Model provides an abstract representation for Web services, their impact on the state of the world and message transmission among them. It is based on the Process Specification Language [Grüniger and Menzel 2003], a formally axiomatized ontology that was originally developed to enable sharing of descriptions of manufacturing processes. Service composition is modeled by complex activities using a set of control constraints (similar to OWL-S control constructs), ordering constraints (specifying just the order of execution), occurrence constraints (linking constraints with a specific activity occurrence)

and state constraints (associating triggered activities with states).

SWSO is based on the situation calculus, which was initially defined with the explicit purpose of solving the frame problem. Hence, Reiter's solution to the frame problem [Reiter 1991] could be applied to SWSF service descriptions. However, the same does not apply to the ramification and qualification problems, which should still be considered unsolved for SWSF descriptions.

### 2.3.6 The SAVVY-WS Framework

SAVVY-WS (Service Analysis, Verification and Validation methodology for Web Services) [Bianculli et al. 2008] is a framework developed by University of Lugano in collaboration with Politecnico di Milano, providing an integrated approach for design-time and run-time validation. The feature that is most interesting to the discussion in this document is the fact that SAVVY-WS relies on the existence of rich specifications for services participating in a composition, and assumes that services are only known through such specifications.

The language use for service specifications in SAVVY-WS is ALBERT [Baresi et al. 2007]. ALBERT is defined over a timed state world, an infinite sequence of states which include some variables and a timestamp, representing a snapshot of a service process. ALBERT assertions are implicitly assumed to be invariants, holding in all states, although we can express the fact that assertions must hold when an event happens using a special predicate. Apart from traditional logical connectives (including universal and existential quantifiers), the timed aspects require the additional definition of temporal operators. Both functional and non-functional properties of services can be expressed using ALBERT.

While SAVVY-WS represents one of the few frameworks that acknowledge the indispensable value of specifications in service-related activities, it still does not explore any of the issues raised in this chapter. No mention is made concerning the existence of the frame, ramification and qualification problems. Moreover, while ALBERT assertions can be expressed to describe what is expected from a

composite service (using so-called guaranteed assertions), actual composite service specification is not provided.

### 2.3.7 Unified Service Description Language (USDL)

USDL [Kadner et al. 2011, Oberle et al. 2013] is the most recent W3C-endorsed effort at defining a service description language and is based on the concept that all aspects of a service, be it business, operational or technical information, need to be expressed in a uniform way. To that end, USDL is defined as a collection of modules, defining specific service aspects on top of a unifying foundation. The modules share a common vocabulary, which is expressed as Linked Data in the most recent version, Linked-USDL. A brief analysis of the USDL modules follows.

**Service** Focuses on the essential structure of a service, describing manual, semi-automated and fully automated services using dependency models.

**Participants** Models the organizational actors that are involved in the provisioning, delivery and consumption of a service.

**Functional** Captures the service functionality at an abstract level, essentially containing IOPE sets as well as exception modeling similar to WSDL faults.

**Interaction** Specifies the external behavioral aspects of the service in the form of protocols and roles.

**Technical** Describes service access mechanisms, via either operation-based or resource-based interfaces.

**Pricing** Defines price plans and models that are used to charge consumption of service functionality.

**Service Level** Provides a way to model Service Level Agreements by incorporating arbitrary attribute and expression languages.



**Legal** Specifies the terms of usage of the service in terms of legal certainty and compliance.

**Foundation** Contains all general elements that are common throughout all other modules.

USDL is successful at providing a unified model for specifying all aspects that may be related to any kind of service, whether it is a Web service, a RESTful service, a business service or even a completely manual service (a human task). However, the Functional and Technical modules, which are the ones more closely related to the goals of this dissertation, make no mention of the representational issues in relation to the frame, ramification and qualification problems. Nevertheless, the modularized and all-encompassing nature of USDL favors the integration of any service description effort, meaning that the language defined in Chapter 3 could be integrated under the USDL umbrella (see also Section 6.3.4).

### 2.3.8 Conclusion

Before completing the review on service description and specification, we offer a comparative summary of the languages presented, focusing on their relation to the frame, ramification and qualification problems. The common conclusion is that none of these languages is capable to express service specifications in the way we have envisioned them. WSDL and SAWSDL, in particular, offer interface-only descriptions, disregarding any information about preconditions and postconditions.

From the rest of the languages presented in this section, OWL-S is the most mature one and the most commonly used in service discovery and composition research approaches. OWL-S, however, does not come without its drawbacks, as it is pointed out in [Balzer et al. 2004], which prevent it from being used in practical real-world scenarios, such as the lack of support for asynchronous services. With regard to the representation problems we examine, there is no mention of them on any research line associated with OWL-S.

WSMO has also been used in many research approaches, even though it has not been around as long as OWL-S. Research by the WSMO group was the first to acknowledge the existence of the frame problem in service description, without, however, proposing any effective solutions. SWSF took things one step further and relied on a formalism, the situation calculus, for which a solution to the frame problem has already been presented and proven. Neither WSMO nor SWSF deal with any other representation problem.

While SAVVY-WS also does not move things forward with regard to the frame, ramification and qualification problems, it is one of very few service research efforts that makes a distinction between simple service descriptions and complete service specifications as well as pointing out the benefits of the latter in comparison to the former. Finally, as the most recent effort, USDL attempts to borrow inspiration from all past research lines in service description; unfortunately, it also inherits their common flaw of disregarding the frame, ramification and qualification problems and their effects in any service-related activity, including, but not limited to service description, discovery, composition and verification.

## **2.4 Automated Service Composition**

We continue the literature review on topics related to this thesis by presenting a comparative analysis of the most representative efforts in automated service composition. We first identify a series of requirements that need to be met, followed by a complete state-of-the-art of automated service composition, organized in four major categories of approaches. At the end of the section, a comparison of the approaches is attempted, based on the extent each approach satisfies the requirements we have identified.

Approaches to the automated service composition problem have been exceptionally diverse and offer different interpretations of what should be addressed in a composition approach. They also differ on the degree of automation that is involved in the process, ranging from semi-automated to fully automated ap-

proaches. Since our survey focuses on automated service composition, any purely manual approaches are omitted.

Due to the multitude and diversity of existing research approaches, it is imperative that we perform some kind of grouping in order to make presentation of the approaches easier. Composition approaches are organized in the following categories:

1. **Workflow-based:** approaches that exploit knowledge from workflow research
2. **Model-based:** approaches that use modeling languages (Petri-nets, UML, FSMs) to represent service compositions
3. **Logic-based:** approaches that use various forms of logic, calculi and algebras
4. **AI planning:** approaches that handle service composition as an AI planning problem

Note that many approaches fit into more than one groups, such as approaches that realize planning using logic-based techniques. In such cases, they are included in the category that represents their primary characteristics; for instance, all planning techniques are placed in the fourth category.

### 2.4.1 Requirements

A considerable number of surveys on service composition (automated or not) have been conducted and published by various researchers all over the world, the most important of which are the following: [Koehler and Srivastava 2003], [Milanovic and Malek 2004], [Rao and Su 2004], [Hull and Su 2005], [Dustdar and Schreiner 2005], [Küster et al. 2005], [Agarwal et al. 2008] and [Marconi and Pistori 2009]. Most of these surveys do not state clearly what requirements need to be met for an approach to successfully solve the problem of automated service

composition and none of them contains a comprehensive comparison based on said requirements. In this section, we present and briefly analyze a set of requirements that must be satisfied in order for a composition process to be considered successful, accurate and complete.

### **Representation completeness**

The first requirement refers to the discussion so far in this chapter. The composition process must rely on specifications that are representationally complete, in the sense that they take into account the frame, ramification and qualification problems and act as a safeguard against the effects of these problems. Service compositions can benefit from such specifications in three complementary ways. First, since participating specifications are free of the frame problem, the resulting composition shares the same characteristic, meaning that non-effects are effectively modeled. Second, the resulting composition takes into account ramifications of effects of the contained services, which may also affect composability results, and protects from inconsistent compositions where a service ramification contradicts a precondition or postcondition of another service in the same process. Third, through a solution to the exogenous qualification problem, explanations can be provided for any unexpected execution result of the composite process.

### **Automation**

Since the focus of this analysis is automated service composition, an obvious requirement is that the generation of the composition schema must be at least partially (if not fully) automated. The main purpose behind designing an approach to handle service composition is to decrease user intervention and accelerate the process of producing a composite service that satisfies preset requirements. Automation reduces the time spent in order to create a composition schema compared to the time required in a manual composition approach, eliminates human errors and reduces the overall cost of the process. Thus, one should expect

that a successful service composition approach provides the highest possible level of automation.

### **Dynamicity**

A defining characteristic of a service composition approach is whether it produces a static or a dynamic composition schema. A static composition approach involves selecting the component services to be used, linking them together to form a composite process, which is then deployed. On the other hand, dynamic composition approaches produce an abstract composition schema, that is essentially a composite process without actual service bindings. The abstract composition schema can be concretized afterwards, when each abstract task in the schema is bound to an actual service endpoint.

Dynamicity ensures that a produced composition schema is consistent and executable long after its initial design. A dynamic composition schema is able to overcome issues such as services no longer being provided and services being replaced by new ones by providers, which would render a static composition schema invalid, inconsistent and impossible to execute. Relying on services described by specifications instead of implementation-specific interfaces facilitates dynamicity, since compositions are essentially collections of specifications and not actually implemented services.

Note that, usually, abstract compositions are produced at design time, with concretization either also taking place at design time or performed at runtime, after execution has begun. This does not necessarily mean that only runtime composition approaches are dynamic; if the result of a design time composition approach is an abstract process not linked to any actual service implementations, then it is also considered dynamic.

### **Semantic capabilities**

As it was argued in Section 2.3, semantic capabilities are fundamental for the realization of SOAs. Semantically rich descriptions of services and composition

goals can be utilized by applications or other services without human assistance, facilitating automated composition. Moreover, they allow for more efficient service matchmaking based not only on service inputs and outputs and, as a result, produced composite services are more close to what the user requests. Thus, effective service composition approaches should exploit semantic descriptions to realize their goals.

### **QoS-Awareness**

QoS-aware approaches take into account not only functional characteristics of services but also non-functional ones, dealing with quality aspects such as response time, price, availability and so on. Considering QoS aspects when deciding which services to include in a service composition schema is important when functional requirements are satisfied by more than one service. As a result, composite services produced by QoS-aware approaches not only offer the capabilities requested by the user but also guarantee the best possible quality. The composition problem of the running example requires QoS-awareness, since it contains non-functional goals both for parts of the process as well as the overall composition.

### **Non-determinism**

Non-determinism involves cases where an action may lead to more than one different states depending on the values of some parameters. For the case of service composition, non-determinism is observed when the composition schema includes choice constructs (such as if-then-else) or loops (such as repeat-while). Non-determinism may lead to an increase in the number of possible choice points throughout a composition problem solution and needs to be taken into account in order to allow for more elaborate composition schemas. For instance, the composition problem in the running example depends on the existence of conditional constructs in order to realize the report delivery sub-process.

### **Partial Observability**

Real-world applications bring several requirements to the table that are not obvious when examining the composition problem from a theoretical point of view. One of these requirements is partial observability, which involves dealing with incomplete information of world states (initial or otherwise). This requirement is strongly linked to approaches that are based on AI planning techniques, since most of these techniques rely on states and transitions to model service behavior. An effective composition approach must deal with incomplete (or in some cases wrong) information and manage to produce composite services despite that fact. For instance, in the running example, we would like to be able to generate composite processes, even if we do not know the exact state of the world at the beginning of the process.

### **Scalability**

Another requirement brought by real-world applications is scalability. The fact that a composition approach works well given a set of services is no guarantee that it will work as effectively with a different, larger or more complex set of services. Composition approaches must be tested against increasingly large service registries or increasingly complex composition problems to examine how their performance is affected. The common trade-off of scalability versus performance often arises in this context. It is important to identify parts of the approach that may pose limitations and try to work them out in order to ensure maximum scalability for a given performance or vice-versa.

### **Correctness**

Composition correctness is required when we want to check if certain properties of the produced composite service hold, such as the fact that it is guaranteed to produce a certain set of outputs given a certain set of inputs and conditions. Correctness is established through verification techniques, which have also been

well researched in service science. Employing service specifications is again a facilitator in this case, because the service itself is associated with the specification against which correctness is checked.

### **Domain independence**

A composition approach should not be limited to a specific domain (unless of course, the focus of the research is exclusive to that domain from the start). One should be able to apply the same approach to different domains, allowing for the solution of a broad range of problems. This requirement may be difficult to achieve while at the same time achieving the requirement of semantic capabilities, since, in some cases, semantic knowledge is domain-dependent.

### **Adaptivity**

The last requirement that we examine has attracted considerable interest from research communities and involves the ability of a service (atomic or composite) to adapt itself. Adaptation is the process of modifying SBAs in order to satisfy new requirements and to fit new situations dictated by the environment on the basis of predefined adaptation strategies. Adaptation goes one step further from dynamic composition approaches, in the sense that the former also deals with changes in the requirements set by the requester which the latter cannot handle.

Adaptation can be proactive, aiming to modify an application before a deviation occurs during the actual operation and before such a deviation can lead to problems; in contrast, reactive adaptation handles faults and recovery from problems reported during execution. While we include adaptivity as a requirement for automated service composition, this does not imply that it should be set as a research goal for such an approach. Instead, research results from the adaptation domain can be employed, so that the resulting composite services have the ability to adapt themselves.



### 2.4.2 Workflow-based Approaches

Business processes and workflow management systems have been a major research topic since the early 1990s. As a result, there has been a lot of research effort on how to represent a sequence of actions. Drawing mainly from the fact that a composite service is conceptually similar to a workflow, it is possible to exploit the accumulated knowledge in the workflow/business process community in order to facilitate service composition. Composition frameworks based on workflow techniques are chronologically one of the initial solutions proposed for automated service composition. Initially, most works focused on static and manual compositions. More recent efforts, however, have attempted to realize automation and dynamicity. Most approaches in this category employ the Business Process Execution Language (WS-BPEL) [OASIS 2007].

[Majithia et al. 2004] presents a framework that automatically constructs a Web service composition schema from a high-level objective. The input objective is fed to an abstract workflow generator that attempts to create an abstract workflow (written in BPEL) that satisfies the objective on one of two ways: either by using already generated workflows or subsets of them that are stored in a repository, or by performing backtracking to find a chain of services that satisfies the objective. The abstract workflow is then concretized, either by finding existing services through a matchmaking algorithm that matches inputs and outputs and binding them to the workflow, or by recursively calling the abstract workflow generator if no service can be found for an activity.

PAWS [Ardagna et al. 2007] is a framework developed by Politecnico di Milano focusing on the adaptation and flexibility of service compositions modeled as business processes. As illustrated in Fig. 2.4, designers create a BPEL process which is then annotated with global and local constraints that usually refer to QoS aspects. The constraints are expressed as Service-Level Agreements (SLAs). For each task in the created process, an advanced service retrieval module attempts to find services that have the required interface (expressed in WSDL or

SAWSDL) and do not violate any constraints, by performing SLA negotiation. If no exact interface matches are found, a mediator is used to reconcile the interface discrepancies. For each task, more than one candidate services are selected. When the process is eventually executed by the BPEL engine, one candidate service is invoked for each task. PAWS also supports self-healing, allowing for faulty services to be substituted by other candidate services and at the same time enabling recovery actions to undo the results of the faulty services.

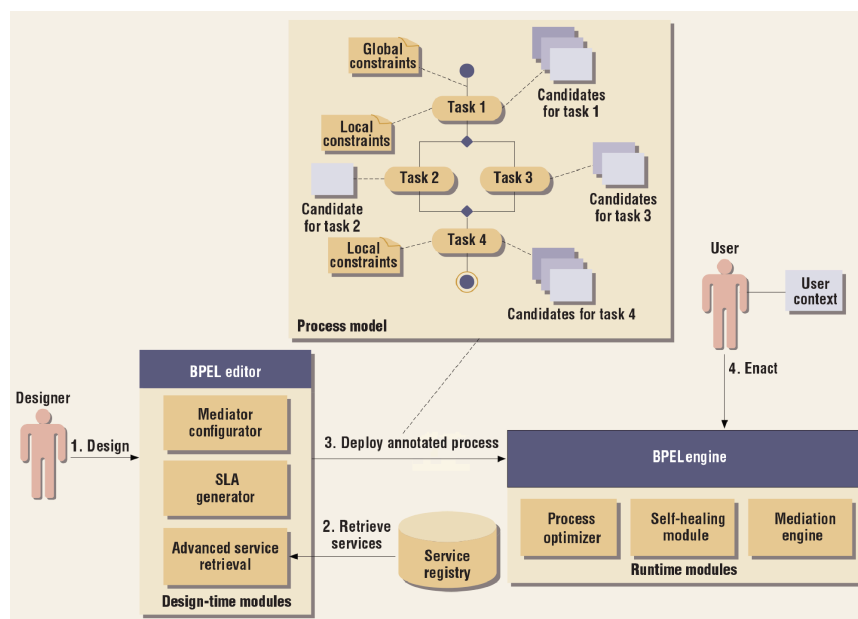


Figure 2.4: The PAWS framework [Ardagna et al. 2007]

[Zisman et al. 2007] approaches the workflow composition problem from a datatype matching perspective, attempting to form sequences of services given a set of inputs, a desired output and a maximum sequence length. Candidate service operations are matched based on linguistic similarity, while inputs and outputs are matched by creating datatype graphs and determining isomorphic relationships between the graphs. The result of this process is one or more sequences of services that achieve the goal output and which are then examined by the service designer in order to determine the most suitable one. This approach is limited to sequences and interface matching, thus not supporting any other control con-

struct (e.g., parallel or conditional composition) or behavioural characteristics such as preconditions and effects.

[Fujii and Suda 2006; 2009] propose an architecture that realizes semantics-based, context-aware, dynamic service composition based on CoSMoS, a semantic abstract model for both service components and users. Their composition approach, named SeGSeC, involves receiving a natural language request from the user which is parsed using preexisting natural language analysis technologies into a CoSMoS model instance. This is fed to the workflow synthesis module which creates an executable workflow by discovering and interconnecting components based on the request and functional descriptions of available components. The workflow synthesis module is apparently limited to sequential and parallel composition schemas. Then, a semantic matching module ensures that the selected components are semantically equivalent to the user request. If more than one components satisfy both functional and semantic properties for a given task, context information is exploited, based on user-defined rules or a history of previous decisions, in order to select the most suitable component. The final workflow is then executed and monitored. When a service failure is detected or a change in context is perceived, the workflow can be dynamically modified to adapt to these changes.

[Pino and Spanoudakis 2012] focuses on the security aspects of workflow composition, augmenting known workflow patterns with security properties and dependencies between inputs and outputs, modeled using situation calculus axioms. Workflow patterns are expressed using the OWL-S process model, although focusing only on its interface subset (inputs and outputs). The composition process is realized in a stepwise manner, identifying the patterns that are applicable and attempting to build a workflow by instantiating the activities in the selected patterns. During instantiation, candidate services are discarded if they fail to satisfy the predefined security properties.

In general, we can conclude that while workflow-based composition approaches have evolved from offering only manual and static composition methods to sup-

porting automation and dynamicity, the resulting workflows are limited to simple schemas such as sequential and parallel execution, or in other cases, such as PAWS, automation is only supported during the execution and adaptation of the workflow, while the workflow design process is manual. This deficiency has been addressed by combining workflow-based methods with AI planning techniques. We examine such works in Section 2.4.5.

### 2.4.3 Model-based Approaches

Model-based or model-driven composition follows the suggestion of system theory to raise the level of abstraction in order to deal with the increasing complexity of systems. Approaches in this category use already explored and well-established models to represent services and service composition, thus using a higher description level on top of the traditional service description in WSDL, OWL-S or similar description frameworks.

E-Service Composition (ESC) [Berardi et al. 2005b] is a prototype tool that implements a model-based technique for automated service composition using Finite State Machines (FSMs). Service behavior is modeled using two schemata expressed as FSMs, an external schema that specifies its exported (externally-visible) behavior and an internal schema that contains information on which service instance executes each given action that is part of the overall service process. When attempting to synthesize a composition, the external FSM models of the available services and the target service are transformed to modal logic formulas in Deterministic Propositional Dynamic Logic (DPDL). If the resulting set of formulas is satisfiable, then a FSM for the target service is produced and converted to an executable BPEL process. In parallel, [Berardi et al. 2005a] presents Colombo, an alternative framework to ESC that is based on transition systems instead of FSMs and focuses on supporting non-determinism and Semantic Web services expressed in OWL-S, as well as introducing the so-called conversation model, where any communication between services is in the form of message

passing. Finally, [Giacomo et al. 2013] revisits this line of work in order to extend it to support partial descriptions of service behaviors and also employs simulation techniques to virtually compute all possible compositions.

Service Aggregation Matchmaking (SAM) [Brogi and Corfini 2008] is a service discovery and composition system based on Consume-Produce-Read (CPR) Nets, a simple variant of the standard Condition/Event Petri Nets defined by the authors to properly model control flow and data flow of a service. The SAM framework can handle both functional and behavioral requests: functional analysis relies on a set of requested functionalities, while behavioral analysis can generate a composite service based on a CPR Net that describes the requested behavior. SAM supports OWL-S services and a translator to CPR Nets is provided. Service composition is realized based on CPR Nets composition, which is formally defined by the authors.

[Tang et al. 2011] proposes a composition approach that combines logic-based and model-based characteristics. Service input/output schemas and behavioral constraints are represented as Horn clauses and service composition is realized through logical inference of these clauses. Composite services are determined through structural analysis of Petri nets that model the Horn clause set and the composition goal. The resulting compositions, however, seem to be restricted at best to sequences of parallel executions.

#### 2.4.4 Logic-based Approaches

This category encompasses all approaches that are based on mathematical foundations such as various logics, calculi or algebras but do not fit in any other category. One example is the use of the pi-calculus [Milner 2004] in [Milanovic and Malek 2004] to describe and compose Web services. Processes in the pi-calculus can be sequences of other processes, parallel compositions, recursive executions, or choices between operations, thus it is possible to express all basic composition schemas. Receiving and sending information between processes is formalized as input and output artifacts exchanged on channels. One can also introduce types

to inputs and outputs. The abstract descriptions that are offered by pi-calculus can be exploited to verify correctness and other properties of compositions.

[Rao et al. 2004] proposes the use of Linear Logic for automated service composition. Linear Logic is an extension of classical logic to model the notion of state evolution by keeping track of resources. The authors propose an automatic translation of OWL-S descriptions to sets of Linear Logic axioms. Then, they use theorem proving to produce the composite service. The target service is expressed as a sequent in Linear Logic and the prover generates all possible compositions which are then translated to process models using a process language inspired by pi-calculus constructs. It is possible to further translate such models to BPEL workflows. However, the authors themselves acknowledge that using a propositional subset of Linear Logic limits the presentation of Web service properties.

[Lécué et al. 2008b] exploits the fact that OWL-S is based on Description Logics and attempts to use DL reasoning to realize service composition. In previous works, the authors defined causal links as semantic links between inputs and outputs of services, ranging from exact matches to disjoint sets (when input and output sets are incompatible). Given a set of services, a causal link matrix (CLM) can be constructed, containing all possible causal links. The composition approach employs CLM+, an extended matrix that also supports non-functional properties. Service discovery is performed to find a set of candidate services based on a request, by calculating the CLM+ for these services and attempting to incrementally achieve the requested output, starting from the given input. At each step, the decision is based on matches inferred from CLM+ matrices between the output of the last service selected and the inputs of all candidate services. The flow diagram of the composition algorithm is shown in Fig. 2.5. [Lécué et al. 2008a] advances the previous work by integrating CLM+ into an extended Golog interpreter (also used in approaches included in the AI planning category in Section 2.4.5) that can compute conditional Web service compositions and can elaborate a strategy for automated branching by means of causal links and laws.

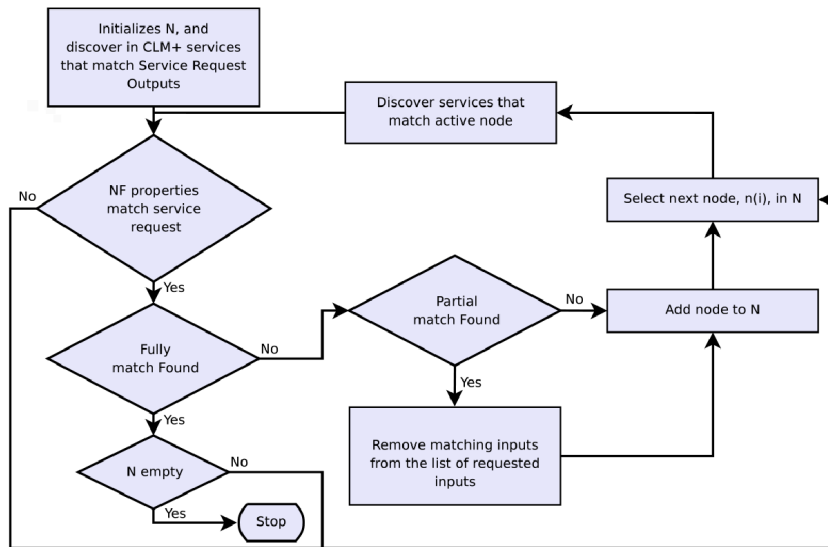


Figure 2.5: Composition algorithm of [Lécué et al. 2008b]

### 2.4.5 AI Planning Approaches

The final category of automated service composition approaches comprises all research efforts that use AI planning techniques in order to generate a composition schema. AI planning techniques involve generating a plan containing the series of actions required to reach the goal state set by the service requester, beginning from an initial state. All approaches in this family rely on one of the many planning techniques that the AI community has proposed and incorporates it in the process model creation phase of the composition framework.

An AI planning problem (in the classical sense) can be described as a quintuple  $\{S, s_0, G, A, \Gamma\}$  [Carman et al. 2003], where  $S$  is the set of all possible states of the world,  $s_0 \in S$  denotes the initial state of the planner,  $G \subseteq S$  denotes the set of goal states the planning system attempts to reach,  $A$  is the set of actions that can be performed in order to reach a goal state and  $\Gamma \subseteq S \times A \times S$  is a transition relation that describes the resulting state or states when a particular action is executed in a given world state. If we consider  $A$  to be the available services,  $G$  to be the goal set by the requester, and a state model related to  $S$ ,  $s_0$  and  $\Gamma$

and applied to the available services, then we can use existing solutions to the AI planning problem in order to solve the service composition problem. It should be noted that this correlation is not followed to the letter by all approaches in this category.

Due to the large amount of approaches that fall into the AI Planning category, we use a further categorization, based on the type of AI planning technique used. The classification is inspired by the works of [Chan et al. 2006] and [Ghallab et al. 2004]. Three categories of planning techniques are examined:

1. **Classical planning:** state-space or plan-space planning
2. **Neoclassical and HTN planning:** graph-based planning, Hierarchical Task Network planning and constraint satisfaction
3. **Other planning techniques:** planning based on the situation and fluent calculi and model checking

### **Classical planning**

Classical planning approaches are based on the definition given at the beginning of this section and involve searching a state-space in order to find a series of state transitions from an initial state to a goal state. Classical planning sometimes involves decomposing a goal to sub-goals and generating a separate plan for each sub-goal, a technique known as plan-space planning.

[Akkiraju et al. 2004] is one of the first efforts to introduce planning in traditional workflow-based service compositions. Given an abstract BPEL flow (with no concrete services bound to the tasks), the goal is to use planning techniques to concretize the workflow. To that end, OWL-S service descriptions are translated to the Planning Domain Definition Language (PDDL) [Ghallab et al. 1998], an effort to standardize planning domain and problem description languages. Then, the PDDL descriptions are fed to IBM's Planner4J planning framework, which contains many planning algorithms, including classical planning ones. At run-time, a



concrete service is requested for each abstract task. If no concrete service found, the planner attempts to solve a planning problem with the available services as input. This approach is semi-automatic, since the creation of the abstract flow is manual.

[Zeng et al. 2008] proposes the formulation of service composition as a goal-directed planning problem that takes three inputs: a set of domain specific service composition rules (which are manually defined), a description of a business objective or goal, and a description of business assumptions (organizational rules and structures). To that end, they devise an ontology to represent these concepts, as well as a three-step composition schema generation process. First, in the Backward-Chaining phase, the composition rules are exploited trying to create a chain starting from the business objective and moving backwards, until there are no more rules or the initial state is reached. Forward-Chaining then attempts to complete the composition schema produced in the first phase by adding services that may be required by the results of some tasks. The final phase, Data-Flow-Inference, adds data flow to the composition schema, since the previous steps only contribute to the control flow aspects of the composition.

[Hatzi et al. 2012] adopts the common method of translating OWL-S descriptions to the PDDL domain, but offers enhanced semantic capabilities through the PORSCE II framework that computes hierarchy relationship and semantic distances in order to determine semantic similarity between OWL-S concepts. Planning is realized through VLEPPO, an integrated system that visualizes planning problems and integrates several different planning algorithms, allowing the user to experimentally explore various solutions. The architectures of PORSCE II and VLEPPO are shown in Fig. 2.6.

### **Neoclassical and HTN planning**

Neoclassical planning comprises techniques that extend the classical notion of planning. These include graph-based planning, where a plan graph of all possible states and transitions (or a subset of those) is constructed and constraint

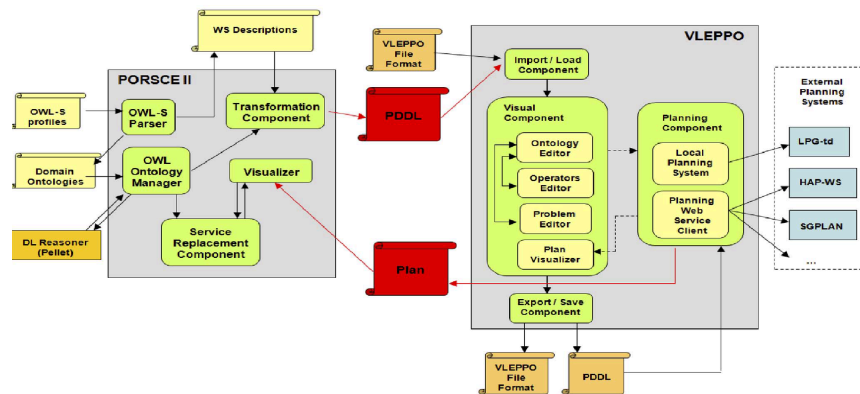


Figure 2.6: Composition architecture of [Hatzi et al. 2012]

satisfaction, where the planning problem is translated to a set of constraints and a solver attempts to find a model that satisfies all constraints. This category also includes Hierarchical Task Network (HTN) planning which involves decomposing the desired task into sub-tasks in a recursive manner, until the resulting sub-tasks can be satisfied.

Graph-based planning is employed by [Wu et al. 2007] in order to realize service composition. The authors propose their own abstract model for service description which is essentially an extension of SAWSDL to more resemble OWL-S and WSMO. In addition, they model service requests and service compositions with similar semantic artifacts. Then, they extend the GraphPlan algorithm so that it can work with the models they defined. Moreover, they add limited support for non-determinism, by detecting patterns that correspond to loops but only in case they are identified beforehand. The final system takes a user request defined using the same model as services and extracts an executable BPEL flow, as shown in the architecture in Fig. 2.7.

[Beauché and Poizat 2008] proposes the use of GraphHTN, which extends GraphPlan with HTN task decomposition in a composition framework focusing on adaptation capabilities. They introduce original semantic structures to describe the capabilities and data that are involved in a service. These structures enable not only vertical adaptation by exploiting semantic relations between capabilities and re-

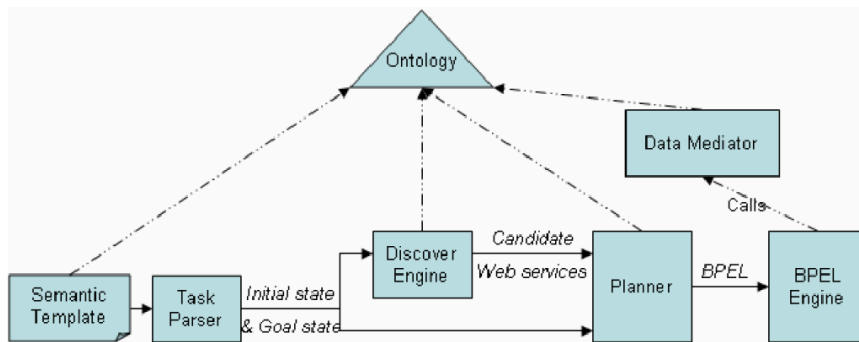


Figure 2.7: Composition architecture of [Wu et al. 2007]

placing one service with an equivalent one, but also horizontal adaptation, substituting missing data with semantically equivalent ones. The final generated plan is transformed to a YAWL [van der Aalst and ter Hofstede 2003] workflow.

[Sohrabi and McIlraith 2009] introduces preferences and regulations to HTN planning. The authors extend PDDL to support preferences over how to decompose tasks as well as expressing preferences over the preferred parametrization of a task. They also include regulations (verification-gearred constraints such as safety constraints) that should be followed by any generated plan. They introduce a modified HTN planning algorithm, HTNWSC, which takes into account both preferences and regulations in the plan generation procedure.

[Mabrouk et al. 2009] addresses QoS-aware service selection and composition by applying clustering and constraint-based techniques eventually producing multiple solutions to support dynamic environments. The framework collects functional and QoS requirements, then discovers all service candidates that satisfy functional requirements, as well as local (task-specific) QoS constraints. Then, a second filtering phase follows, based on global QoS constraints, essentially maximizing a utility function. At runtime, a unique service out of the ones not filtered out is selected for each task in the composition. As pointed out in Section 2.4.6, this work is one of the most successful with regard to satisfying the requirements in Section 2.4.1, only failing to support partial observability as well as not concerning itself with correctness methods.

[Rosenberg et al. 2009] presents a novel composition language called VCL that is the foundation of a framework that realizes composition using graph-based techniques and QoS-aware optimization using constraint and mixed-integer programming techniques. Requirements are expressed in VCL in the form of constraints and candidate services are selected from a registry. A structured composition is produced based on a dependency graph that analyzes the required data flow. The most suitable services are selected from the candidate ones by either solving a constraint satisfaction problem for both local and global QoS requirements or applying integer programming techniques. [Bartalos and Bieliková 2012] also employs directed acyclic graphs in order to find all possible compositions of services; then, a forward and backward chaining composition algorithm is applied, that takes into account IOPE-based service descriptions and relies on service space restriction to achieve efficiency in case of large repositories and continuous composition requests.

[Barakat et al. 2011] presents a hierarchy-based dynamic composition approach that relies on planning knowledge organized as task hierarchies and decompositions. In contrast to HTN planning, a task at any level of granularity can be mapped to a concrete service. Pruning is performed on both task and plan levels based on QoS constraints to reduce search space and increase performance. Then, to find the global optimal plan, candidate plans are modeled as directed graphs and a multi-constrained optimal path selection problem is solved.

### **Other planning techniques**

[McIlraith and Son 2002] first explored planning using Golog, a logic programming language based on the situation calculus. They extended and customized Golog to support personalized constraints and non-determinism in sequential executions and modified ConGolog, a Golog interpreter to realize these enhancements. This work was developed concurrently with the definition of OWL-S (then called DAML-S) and was one of the first to consider Semantic Web services as an input to planners via translation to PDDL. In a more recent work [Sohrabi et al. 2009]

the introduction of preferences to planning with Golog is proposed, in the same way that [Sohrabi and McIlraith 2009] introduced preferences to HTN planning. Preferences are expressed using a first-order language defined by the authors and used in a modified version of a Golog interpreter. Evaluation results illustrate the effectiveness of introducing preferences to find optimal compositions.

[Chifu et al. 2009] and [Bhuvaneshwari and Karpagam 2010] follow a similar approach but are based on the fluent calculus instead. OWL-S service descriptions are translated to fluent calculus theories and service composition is realized via planning with FLUX, a fluent calculus logic programming language. These works are directly related to this thesis, since both WSSL and the composition and verification framework we propose are based on the fluent calculus. However, in contrast to our work, the simple fluent calculus is used by [Chifu et al. 2009] and [Bhuvaneshwari and Karpagam 2010], without the extensions that solve the ramification and qualification problems, resulting in frameworks that ignore their effects and at the same time fail to capitalize on the benefits of their solutions. Moreover, both works do not support control constructs other than sequential and parallel execution. Finally, their choice to represent inputs and outputs using the extension of the fluent calculus to support knowledge and sensing is invalid since these constructs represent knowledge states and handle partial observability. In our work, inputs and outputs are represented in a natural way by special fluents.

The final planning paradigm that we examine is planning as model checking, which has been extensively explored in correlation with service composition by the ASTRO team in Fondazione Bruno Kessler. The original works in [Pistore et al. 2004, Traverso and Pistore 2004] attempt to exploit planning by model checking in order to deal effectively with non-determinism, partial observability, and complex goals. OWL-S process models are translated to state transition systems while goals are expressed using EAGLE, a requirements specification language. State transition systems and goal descriptions are fed to the MBP planner (which uses model checking) and evaluation shows that, while correct plans are produced, the procedure does not scale well, mainly due to the way goals were expressed.

Subsequent research ([Bertoli et al. 2010, Pistore et al. 2005a;b]) attempts to address scalability issues by defining an appropriate model for providing a knowledge level description of the component services. In these works, BPEL workflows are used as input, instead of OWL-S process models, and are translated to knowledge level models (essentially a set of propositions). Given a composition goal, they automatically generate its knowledge level representation that declares what the composite service must know and how goal variables and functions must be related with the variables and functions of the component services. This new representation of goals causes an increase in scalability.

These research results have been incorporated in the ASTRO framework [Trainotti et al. 2005], allowing it to realize automated service composition. ASTRO takes BPEL processes as input, feeds them into a planner that implements the planning as model checking technique and exports resulting plans in the form of BPEL processes. ASTRO also includes an execution and monitoring component, based on the ActiveBPEL engine.

Other research efforts from the ASTRO team have focused on devising new models for the specification of data flow [Marconi et al. 2006] as well as control flow [Bertoli et al. 2009] requirements. Data flow requirements specify how output messages (messages sent to component services) are obtained from input messages (messages received from component services). Control flow requirements involve termination conditions and transactional issues. Data flow requirements are translated to state transition systems while control flow requirements are translated to the EAGLE language, although previous work has indicated that this caused some scalability issues. [Marconi et al. 2008] shows how these research results are incorporated in the ASTRO framework.

#### **2.4.6 Comparison and Research Challenges**

In this section, we present comparison tables for all approaches analyzed in the literature review. The comparison is based on the requirements that were out-

lined in Section 2.4.1. Publications that form a single line of work are represented by a single entry in the comparison tables.

Tables 2.1 and 2.2 present a comprehensive comparison that details how each approach manages to meet the automated composition requirements that we defined. A ● denotes that the corresponding approach satisfies that particular requirement. A ◐ denotes that the corresponding approach only partially satisfies that particular requirement (e.g., in the case of scalability, evaluation is conducted and shows that scalability is not always achieved). No symbol denotes that the corresponding approach does not satisfy that particular requirement (or the authors do not deal with the requirement at all). We exclude the requirement of adaptivity, since it is the main objective of a multitude of research works focusing primarily on service adaptation rather than composition and thus, is considered out of scope.

Table 2.3 contains a condensed comparison which lists only categories of approaches instead of individual approaches, with the purpose of determining research gaps for each category as well as in general. A ● denotes that the particular requirement has been addressed by the majority of approaches in the category. A ◐ denotes that the particular requirement has been addressed by a single approach in the category. No symbol denotes that no approaches in this category have addressed that particular requirement.

By studying Tables 2.1, 2.2 and 2.3, we infer a series of research topics that still pose a challenge and drive the research that constitutes this thesis. First and foremost, since no service description language solves the frame, ramification and qualification problems, no composition approach achieves the requirement of representation completeness. Hence, no composition approach takes ramifications into account, composition correctness disregards the exogenous qualification problem and the produced composition descriptions do not include specification of non-effects. To make sure that none of the issues analyzed in Section 2.2 come up at any stage before, during or after the composition process, composition approaches must rely on service specifications that take into account the frame,

	Representation Completeness	Automation	Dynamicity	Semantic Capabilities	QoS Awareness	Non-determinism	Partial Observability	Scalability	Correctness	Domain Independence
[Majithia et al. 2004]		●	●	●						
[Ardagna et al. 2007]		◐	●	◐	●	●				●
[Zisman et al. 2007]		●								●
[Fujii and Suda 2006; 2009]		●	●	●				●		●
[Pino and Spanoudakis 2012]		●	●						●	●
[Berardi et al. 2005a;b, Giacomo et al. 2013]		●				●	●		●	●
[Brogi and Corfini 2008]		●		◐				●	●	●
[Tang et al. 2011]		●		●					●	●
[Milanovic and Malek 2004]		◐						●	●	●
[Rao et al. 2004]		●		●					●	●
[Lécué et al. 2008a;b]		●	●	●	●					●
[Akkiraju et al. 2004]		◐	●	●		●			●	●

Table 2.1: Comparison of Automated Service Composition approaches - part 1



	Representation Completeness	Automation	Dynamicity	Semantic Capabilities	QoS Awareness	Non-determinism	Partial Observability	Scalability	Correctness	Domain Independence
[Zeng et al. 2008]		●		●	●				●	
[Hatzel et al. 2012]		●	●	●				●		●
[Wu et al. 2007]		●		●		●			●	●
[Beauchamp and Poizat 2008]		●		●					●	●
[Sohrabi and McIlraith 2009]		●		●				●	●	●
[Mabrouk et al. 2009]		●	●	●	●	●		●		●
[Rosenberg et al. 2009]		●	●		●			●		●
[Bartalos and Bieliková 2012]		●	●		●			●		●
[Barakat et al. 2011]		●	●	●	●			●		●
[McIlraith and Son 2002, Sohrabi et al. 2009]		●		●		●		●		●
[Bhuvaneshwari and Karpagam 2010, Chifu et al. 2009]		●		●	●					●
[Bertoli et al. 2010, Pistore et al. 2005a;b]		●		●		●	●	●	●	●

Table 2.2: Comparison of Automated Service Composition approaches - part 2

	Representation Completeness	Automation	Dynamicity	Semantic Capabilities	QoS Awareness	Non-determinism	Partial Observability	Scalability	Correctness	Domain Independence
Workflow-based		●	●	●	◐	◐		◐		●
Model-based		●		◐		◐	◐	◐	●	●
Logic-based		●	◐	●	◐			◐	●	●
Classical planning		●	●	●	◐	◐			●	●
Neoclassical/HTN		●	●	●	●			●	●	●
Other planning		●		●		●	●		●	●

Table 2.3: Comparison of Automated Service Composition approaches - by category

ramification and qualification problems. We present WSSL, a language designed with the explicit purpose of addressing these problems in Chapter 3.

Non-determinism and partial observability are the two requirements addressed by the least number of approaches: only [Brogi and Corfini 2008] and [Bertoli et al. 2010] propose composition approaches that satisfy them both simultaneously. However, both of them are indispensable for an approach to be considered useful in real-world applications. Composition schemas, may consist mostly of sequential and parallel executions but conditional execution is also a very common behavior that needs to be modeled, as is iteration. Also, assuming that we have complete knowledge of the state before executing a service composition may be suitable for theoretical approaches, but in actual scenarios it is crucial to be able to work with states that are only partially observable. In Chapter 4, we extend WSSL in order to support both deterministic and non-deterministic composition constructs, as well as modelling partially observable states.

While earlier approaches to service composition addressed almost exclusively functional composition, later works began making efforts to realize QoS-awareness

by introducing or reusing existing quality models, quality aggregation methods and satisfaction of both functional and non-functional goals. Since QoS-aware composition is not as widely researched as its functional counterpart, there are several aspects that are yet to be fully achieved. First, the QoS models employed are usually incomplete. As a result, the proposed QoS aggregation methods either do not take into account some QoS attributes or they focus exclusively on specific ones, resulting in a specialization of the aggregation process, as is the case of [Rosenberg et al. 2009], which contains the most complete aggregation model in terms of supported composition patterns.

Apart from the aggregation process, QoS-aware composition usually relies on planners that produce a single abstract plan (in almost all cases sequential) that is then concretized based on QoS constraints (as in [Mabrouk et al. 2009] and [Rosenberg et al. 2009]); in other cases more than one plans are produced and concretization is performed for all of them (as in [Barakat et al. 2011]). In the first case, there may be possible plans that produce more optimal solutions after concretization but are ignored, while in the second case, efficiency is compromised, especially if the number of plans rises significantly. In Chapters 4 and 5, we address all these deficiencies, by first extending WSSL to support quality specification based on [Kritikos and Plexousakis 2009b], which contains arguably the most complete analysis of quality attributes; then, we design and implement WSSL/CVF, a complete QoS-aware service composition and verification framework relying on WSSL. The primary objective for the proposed framework is not only to achieve QoS-awareness and support non-determinism and partial observability, but to satisfy all automated composition requirements that are included in Section 2.4.1.

## **2.5 Specification-based Service Discovery**

The proposed framework relies on WSSL not only to realize service composition but also to implement specification-based service discovery and verifica-

tion. This and the following section provide a brief analysis of the most related approaches in these fields. In both cases, we mainly focus on works that rely on rich service specifications, containing conditions before and after execution and possibly supporting semantics.

In [Rao et al. 2006], the authors argue that full automation for service discovery and composition is unrealistic and propose a mixed initiative approach, combining human decision making and partial automation. Services are specified using OWL ontologies, relying on a simplified version of OWL-S [Martin et al. 2004], supporting IOPE representations and also allowing for the inclusion of non-functional properties (although this is not actually realized in this work). Discovery is realized via reasoning based on semantic rule engines. Evaluation shows a significant overhead in loading service description ontologies.

Service Aggregation Matchmaking (SAM) [Brogi and Corfini 2008] relies on translating OWL-S services to Consume-Produce-Read (CPR) Nets, a simple variant of the standard Condition/Event Petri Nets defined by the authors to properly model control flow and data flow of a service. Discovery can be both functional, i.e. simple input/output matching based on the notion of subsumption, and behavioral, also including the behavior of services based on OWL-S control flow constructs, not including, however, preconditions and effects. In both cases, discovery is realized via CPR Nets analysis.

In [Klusch et al. 2009] and [Klusch and Kaufer 2009] present two Semantic Web service matchmakers, OWLS-MX and WSMO-MX, with the former relying on OWL-S service descriptions and the latter exploiting WSMO services. Both systems offer a hybrid approach in discovering services, supporting both logic-based filtering based on ontology reasoning and non-logic-based actions, namely syntactic similarity. Logic-based filtering in OWLS-MX yields exact, subsumption and plug-in matches while hybrid filtering also delivers the inverse of subsumption (subsumed-by) and nearest neighbor relations. In all cases, OWLS-MX takes into account only inputs and outputs. WSMO-MX, on the other hand, relies on the complete IOPE specification of a WSMO service and yields more detailed matches

that take into account preconditions and effects expressed using F-Logic. Evaluation results show good performance in terms of precision and recall, but overall computation time is high, around 10 seconds per query for repositories of 500 services.

[Paliwal et al. 2012] proposes an unconventional approach to semantic service discovery, due to the fact that Semantic Web service standards such as SAWSDL, OWL-S and WSMO have not been widely adopted. Instead of relying on them, semantics are derived by parsing WSDL documents in the UDDI registry and attempting to find ontology concepts for each operation, input and output term included in them. Based on these semantics, services are clustered in order to reduce the search space and discovery is performed using semantic similarity matching. On the other hand, [Lemos et al. 2012] propose to extend semantic similarity with quality-based preferences while also supporting both hard and soft constraints. While both of these works possess innovative characteristics, neither goes beyond input/output matchmaking.

[Spanoudakis and Zisman 2010] presents a UML-based discovery framework that aims to assist designers in creating service-based systems. Discovery queries consist of a structural model (UML class diagram), describing the required interface, behavioral models (UML sequence diagram), describing the required business process (similarly to BPEL, not taking into account preconditions and effects) and additional constraints, both hard and soft, e.g., quality-based constraints, defined using ConstraintSQL, an XML-based language defined by the authors. The discovery process consists of the filtering phase, where hard constraints are taken into account and the optimization phase, where an overall distance is calculated based on the structural and behavioral models and soft constraints. Evaluation results show that highest precision and recall is achieved when discovery queries involve both structural and behavioral parts as well as additional constraints; however, computation time is significantly higher when combining all these features, ranging from 26 to 134 seconds for exact matching.

Finally, the more recent work of [Zisman et al. 2013] proposes a proactive

and reactive runtime service discovery framework, called RSDF, that adopts the characteristics of discovery queries presented in the UML-based framework but extends its capabilities beyond service-based system design to runtime discovery. Queries are defined using SerDiQueL, an XML-based language that allows for structural criteria (required interface), behavioral criteria (preconditions, ordering and dependencies between service operations) and additional constraints, referring to quality aspects or non-structural interface characteristics, further characterized as contextual or non-contextual. The discovery process in RSDF is executed in pull or push modes, with the former addressing initial bindings and discovery requests and the latter addressing requests for service replacement. In both cases, evaluation consists of filtering and optimization phases similar to the previous framework. Evaluation shows that computation time is much shorter in push mode, hence it can be used for runtime discovery; this is in contrast to pull mode, which increases linearly with the repository size but requires, 1 second per service on average, when taking into account all aspects of a SerDiQueL query.

### 2.5.1 Discussion

In general, while there exist several approaches that go beyond simple syntactic-based discovery, the majority of them exploit only the semantic aspects of languages such as OWL-S or WSMO, viewing them merely as ontologies for the semantic annotation of service interfaces. Such works are fundamentally inadequate with regard to the goals of this thesis, since they view service description as a simple collection of inputs and outputs, rather than adopt the approach of specifying the complete behavior of a service.

The few works that address these deficiencies ([Klusck and Kaufer 2009, Rao et al. 2006, Zisman et al. 2013]) achieve a more complete approach to the discovery problem, allowing for solutions that take simultaneously into account service interfaces, behaviour and quality. However, they still are unable to address the following issues:

- All approaches disregard the existence of the frame, ramification and qualification problems even though they include conditions in service and discovery query descriptions, resulting in incomplete service behaviour models that may lead to inconsistent or inaccurate discovery solutions, e.g., due to ignoring a ramification of a candidate service or by failing to specify non-effects.
- Computation time seems to be affected by the inclusion of multiple measurements for each facet of a discovery query, due to the fact that there is no unified approach to the specification of all aspects of service behaviour.
- The discovery process is applied on service repositories, assuming each service implementation as a unique entity, further affecting performance, even though multiple implementations of the same functionality can be grouped under a single behaviour specification.

## 2.6 Verification of Service Behavior

The work of [Foster et al. 2003] proposes a model-based approach for service verification, using Finite State Processes (FSP). Service specifications are assumed to be expressed using UML Message Sequence Charts and are translated to FSPs using a preexisting tool. Concrete service implementations are expressed using BPEL and a mapping from BPEL to FSPs is provided. Verification then amounts to creating joint labeled transition systems (LTS) based on the two FSP specifications and checking trace equivalence. The LTSs are restricted to input and output message exchanges between services participating in a composition. Foster et al. [2004] follows up on the previous work by focusing on service choreographies and specifically addresses verification of interface compatibility, safety and liveness properties.

[Fisteus et al. 2004] also focuses on verification of services modeled as BPEL processes using Finite State Machines (FSM); it differs, however, in that FSMs

are, in turn, translated to verification-specific models so that pre-existing model checking verification systems, such as SPIN [Holzmann 2004], can be reused. Verification properties are expressed using boolean predicates and include invariants, goals (i.e. effects), preconditions and postconditions, essentially offering more detailed verification actions, rather than focus on inputs and outputs only.

[Kazhamiakin et al. 2006] proposes a holistic approach that takes into account both synchronous and asynchronous service execution models. Transition systems are again used in order to model service behavior, augmented with channels that model message exchange queues; every communication model, synchronous or asynchronous, is represented by a different channel configuration. State-of-the-art model checkers, such as SPIN [Holzmann 2004], are then used to verify properties expressed in Linear-time Temporal Logic (LTL), specifically termination of individual services and of the composition as a whole.

[Dranidis et al. 2009] proposes the use of Stream X-machines (SXM) as a model for service verification due to their ability to represent both control and data flow. Service behaviour is essentially modeled as a state transition diagram, supporting inputs, outputs, preconditions and effects for each operation. Verification is performed at run-time by monitoring service execution and simulating it at the same time using the corresponding SXM in order to compare actual and simulated results. Although SXM modeling supports preconditions and effects in theory, in practice the authors do not include them in their approach. Also, the process of deriving the SXM model of a Web service is manual.

Similarly to [Foster et al. 2003], labeled transition systems (LTS) are also used by [Sheng et al. 2014] as a way to model service behavior for verification purposes. The authors assume that operational and control behaviors of a service (represented by LTSs) are separated, while a link between them is kept via conversational messages. Verification first involves making sure that both behaviors are synchronized by checking the message sequences. Then, properties are extracted from the control behavior model and are verified against the operational behavior one. Behavior modeling is again limited to input and output messages.



### 2.6.1 Discussion

The vast majority of verification approaches for service behavior, regardless of the models and representations used, focuses on properties that are related to inputs and outputs of services and service compositions, as well as message exchanges and conversations. Given that, modeling service behavior essentially amounts to determining the input and output messages involved and the order in which they are exchanged, disregarding any knowledge about whether an actual message exchange can happen (through precondition modeling) or whether the exchange completed successfully (effect modeling), even though the models used may support such expressions. As was the case with service discovery, such works differ in principle from the foundations and goals of this thesis, which requires, at a bare minimum, for service behavior to include a representation of preconditions and effects.

[Fisteus et al. 2004] is the only notable service verification effort, to the best of our knowledge, that allows for the specification of verification properties that include preconditions and effects. However, such properties are only included explicitly as verification goals and are not part of the service specification language employed. Moreover, the frame, ramification and qualification problems are not taken into account; hence, verification is unable to take into account the complete behavior of a service, including possible knock-on or indirect effects or abnormal and unexpected cases that deviate from the default service behavior.

From the brief literature review on service discovery and verification it should become apparent that the frame, ramification and qualification problems are not solely associated with the description and specification of services, but permeate the full lifecycle of a service or SBA. Hence, creating a service specification language that addresses these problems effectively benefits all tasks that are based on specifications, namely description, composition, discovery and verification. This will become apparent in the design and implementation of the proposed composition and verification framework, WSSL/CVF, in Chapter 5.



## Chapter 3

# Web Service Specification Language (WSSL)

### Contents

---

<b>3.1</b>	<b>The Fluent Calculus</b> . . . . .	<b>71</b>
3.1.1	General Notational Conventions . . . . .	71
3.1.2	Basic Definitions . . . . .	71
3.1.3	Actions, State Change and the Frame Problem . . . . .	73
3.1.4	Representing Inputs and Outputs . . . . .	76
<b>3.2</b>	<b>Abstract Syntax</b> . . . . .	<b>77</b>
3.2.1	Identifiers and Namespaces . . . . .	77
3.2.2	Service Specifications . . . . .	78
3.2.3	Addressing the Ramification Problem . . . . .	82
3.2.4	Addressing the Qualification Problem . . . . .	84
3.2.5	WSSL specification of the running example . . . . .	86
<b>3.3</b>	<b>Surface Syntax</b> . . . . .	<b>89</b>
<b>3.4</b>	<b>Semantics</b> . . . . .	<b>90</b>
3.4.1	Satisfaction and Entailment . . . . .	92
<b>3.5</b>	<b>WSSL/XML</b> . . . . .	<b>93</b>

---

In this chapter, the main contribution of this thesis, the Web Service Specification Language, is presented. Beginning with a detailed analysis of the language's fluent calculus foundations, we then move on to a formal definition of WSSL (pronounced /'wi:zəl/) in terms of its abstract syntax and semantics, focusing on the way the language addresses the frame, ramification and qualification problems. Afterwards, a surface syntax as well as an XML syntax are presented, aiming at increasing human readability and machine interpretability, respectively.

Employing a formalism designed for the domain of Reasoning about Action and Change (RAC) allows us to more accurately specify the effects of a service execution and the state of the world before and after each execution. The rationale behind choosing the fluent calculus over other RAC formalisms such as the situation and event calculi is briefly summarized as follows:

- Service specification requires a non-narrative-based formalism, since there is no need for an explicit notion of time: we only need to recognize the state before and the state after execution. Hence, the event calculus is not a suitable formalism, while the situation and fluent calculi are.
- In terms of complete solutions to the frame, ramification and qualification problems, the fluent and event calculi are the best candidates, due to the work in [Thielscher 2005b] and [Kakas et al. 2011], respectively, as presented in Section 2.2. While the situation calculus was designed specifically to address the frame problem, no adequate solution to the ramification and qualification problems exists, to the best of our knowledge.
- In terms of tool support, all three calculi can be expressed and reasoned with in the form of a logic program. While event calculus reasoning has received more attention in recent years, an implementation of the fluent calculus is offered, in the form of FLUX, and has already been applied in programming autonomous agents, a field closely related to service science.

It is important to stress the fact that WSSL is independent of service design

models. The language design was driven by traditional WSDL-based Web services as well as Semantic Web services, but WSSL specifications can describe the behavior of any service-based system or application, including RESTful Web services [Richardson and Ruby 2007], and can be exposed in a straightforward way as Linked Services, provided that their design principles [Pedrinaci and Domingue 2010] are followed.

## 3.1 The Fluent Calculus

### 3.1.1 General Notational Conventions

Throughout the dissertation document, predicate and function symbols start with a capital letter, while variables start with a lowercase letter. For the sake of simplicity, in definitions, a term  $x$  may also denote a sequence of terms  $x_1, \dots, x_n$ , unless stated otherwise. Fluent variables are written by the letters  $f$  or  $g$ , state variables by the letter  $z$  and situation variables by the letter  $s$ , possibly followed by subscripts. Finally, within all formulas, variables outside the range of quantifiers are implicitly universally quantified.

### 3.1.2 Basic Definitions

The fluent calculus [Thielscher 2005b] is a specification language and system for reasoning about action and change, designed for autonomous robotic agents with the purpose of supporting non-determinism and partial observability, knowledge and sensing actions, ramifications and concurrency. Following an initial definition in [Thielscher 1997], a set of extensions were proposed in [Thielscher 1999; 2001b], resulting in a language that addresses both representational and inferential aspects of the frame problem, as well as the ramification and qualification problems. The fluent calculus, as it is used for the purposes of this thesis, uses standard many-sorted first-order predicate logic, combined with a few customary conventions regarding sorts and equality.

The fundamental entity of the fluent calculus is the *fluent*, a single atomic property of the physical world which may change in the course of time. In the initial definition of the fluent calculus as a specification language for robotic agents, this change is the result of manipulation by the robot. In the case of services, a fluent changes value as a result of a service execution; for instance, the execution of the *RetrieveLocation* service in the running example results, among others, in the addition of the fluent  $Retrieved(location, user)$  to the current state, denoting that the location of the particular user is now retrieved by the system. Note that fluents are represented by functions that take zero or more variables as arguments.

A *state* is a snapshot of the environment at a certain moment. A fluent is equivalent to a state where only this particular fluent holds. An empty state, denoted by  $\emptyset$ , is a state where no fluent holds. Two states can be combined in order to form a new one using state composition, denoted by the function  $\circ$ . *Actions* represent executions of service operations. Finally, a *situation* is a history of action performances. The initial situation, where no actions have taken place, is denoted by  $S_0$ . Predefined function *State* maps a situation to the state of the environment in that situation.

A fluent  $f$  is said to hold in a state  $z$ , if  $z$  can be decomposed into two states, one of which is  $f$ . The macro  $Holds(f, z)$  is introduced for notational convenience:

$$Holds(f, z) \stackrel{def}{=} (\exists z') \cdot z = z' \circ f$$

State composition is governed by the following foundational axioms:

1. Associativity:  $(z_1 \circ z_2) \circ z_3 = z_1 \circ (z_2 \circ z_3)$
2. Commutativity:  $z_1 \circ z_2 = z_2 \circ z_1$
3. Empty state axiom:  $\neg Holds(f, \emptyset)$
4. Irreducibility:  $Holds(f, g) \Rightarrow f = g$

5. Decomposition:  $Holds(f, z_1 \circ z_2) \Rightarrow Holds(f, z_1) \vee Holds(f, z_2)$
6. State equality:  $(\forall f)(Holds(f, z_1) \equiv Holds(f, z_2)) \Rightarrow z_1 = z_2$

*Holds* expressions can be combined to create more complex state description formulas. Such a first-order formula  $\Delta(z)$  is called a *state formula*, if  $z$  is the only free state variable, with states occurring exclusively in *Holds* expressions, without any actions or situations. Accordingly, a *situation formula*  $\Delta(s)$  is produced given a state formula  $\Delta(z)$  and a situation  $s$  such that  $z = State(s)$ .

### 3.1.3 Actions, State Change and the Frame Problem

In order to formalize preconditions of actions, a predefined predicate *Poss* is introduced, taking an action and a situation (or an action and a state) as arguments. Also, predefined function *Do* takes an action and a situation as input and produces the situation that results after performing the action on the input situation. Preconditions are expressed using the following definition:

**Definition 3.1.1** An *action precondition axiom* for an action  $A(x)$  is a formula

$$Poss(A(x), s) \equiv \Pi_A(x, s)$$

where  $\Pi_A(x, s)$  is a situation formula with free variables among  $x, s$ , with the semantics that action  $A$  is possible at situation  $s$  if and only if  $\Pi_A$  is true.

If  $z = State(s)$ , then the action precondition formula can be written equivalently as  $Poss(A(x), z) \equiv \Pi_A(x, z)$ . For instance, the action precondition axiom for the *RetrieveLocation* service of the running example is defined as:

$$Poss(RetrieveLocation, z) \equiv Holds(GPSActive(user), z)$$

The fluent calculus bases its solution to both the representational and inferential aspects of the frame problem on the notion of states. Change is modeled as the difference between two states. Actions are deterministic and result in a bounded number of direct effects. Change can be positive (a fluent is added to a state) or negative (a fluent is removed from a state). Formally:

$$z - f = z' \stackrel{def}{=} \neg Holds(f, z') \wedge (z' \circ f = z \vee z' = z) \text{ and}$$

$$z + f = z' \stackrel{def}{=} Holds(f, z') \wedge (z \circ f = z' \vee z' = z).$$

Change can be generalized to include *finite states*, which are defined as either the empty state, or a composition of a finite number of state terms expressed by function symbols representing fluents.

Given this state change modeling, effects are expressed based on the following definition:

**Definition 3.1.2** A *state update axiom* for an action  $A(x)$  is a formula

$$Poss(A(x), s) \rightarrow (\exists y)(\Delta(s) \wedge State(Do(A(x), s)) = State(s) + \theta^+ - \theta^-)$$

with  $\Delta(s)$  a situation formula with free variables among  $x, y, s$  and  $\theta^+, \theta^-$  finite states with variables among  $x, y$ . The semantics is the following: provided that an action  $A$  is possible at a situation  $s$ , then the action execution at situation  $s$  results in a successor state which is defined as a modification of the previous state ( $State(s)$ ), resulting after adding fluents that have been made true ( $\theta^+$ , called *positive effects*) and subtracting fluents that have been made false ( $\theta^-$ , called *negative effects*), under possible additional conditions expressed by formula  $\Delta(s)$ .

In the simplest case,  $\Delta(s) \equiv \top$ . For instance, the state update axiom for the *RetrieveLocation* service is expressed as:

$$Poss(RetrieveLocation, s) \rightarrow$$

$$(\exists location, request)(State(Do(RetrieveLocation, s)) =$$

$$State(s) + Retrieved(location, user) - HasInput(request))$$

Note that Definition 3.1.2 is a simplified version of a state update axiom, allowing a single possible state update, given an action and a set of additional conditions. The general form of a state update axiom allows a finite number of possible sets of additional conditions for the same action, each resulting in a different state



update as seen below:

$$\begin{aligned} Poss(A(x), s) \rightarrow & (\exists y_1)(\Delta_1(s) \wedge State(Do(A(x), s)) = State(s) + \theta_1^+ - \theta_1^-) \\ \vee \dots \vee & (\exists y_n)(\Delta_n(s) \wedge State(Do(A(x), s)) = State(s) + \theta_n^+ - \theta_n^-) \end{aligned} \quad (3.1)$$

In the case of services, it is far more common to include all required conditions for a successful execution as service preconditions, without separating them in preconditions and additional conditions. Hence, the simplified state update axiom is deemed more appropriate. Under the assumption that  $\theta^+$  and  $\theta^-$  are disjoint, state update axioms are a provably correct solution to the representational aspect of the frame problem (see Theorem 7 in [Thielscher 2000] and Theorem 1.14 in [Thielscher 2005b]).

Definitions 3.1.1 and 3.1.2 also allow the implicit specification of invariants, i.e. conditions that should hold both before and after service execution. Invariant semantics can be realized by including the invariant condition in the action precondition axiom and making sure that the corresponding state update axiom does not include a negative effect that refers to the same condition. For instance, one precondition of the *ReceivePay* service is that the mechanical problem has been solved. Due to the nature of state change in the fluent calculus, not including the same condition in the negative effects of the *ReceivePay* state update axiom can replicate invariant semantics.

States follow the principle of inertia, meaning that the values of the fluents they comprise tend to persist. In the running example, this is manifested as the fact that a credit card remains deactivated after having reached the daily spending limit. A change in the activation state of the credit card occurs either explicitly, as an effect included in another service specification or implicitly, by requiring that the credit card is active as a precondition for other services. In general, there is a limit to what one can observe and specify and it is usually expected to include a parsimonious analysis of change in each service specification.

WSSL follows the SOA design principle of service statelessness: state management is delegated and deferred to other services. Hence, modeling stateful infor-

mation that needs to be kept among executions of a single service is out of the language scope. Concerning internal states of a service, WSSL considers atomic services as black boxes, only recognizing the states before and after execution. In the case of composite services, however, internal states can be modeled, based on the before and after states of each participating service. Section 4.1 details a WSSL extension to support specification of composite services.

### 3.1.4 Representing Inputs and Outputs

While action precondition axioms and state update axioms of the fluent calculus correspond directly to service preconditions and postconditions, the translation is not as direct in the case of inputs and outputs. In related literature ([Bhuvaneshwari and Karpagam 2010, Chifu et al. 2009]) inputs and outputs are modeled using the extension of the fluent calculus to support knowledge and sensing. This is fundamentally wrong, since the intent of these extensions is to represent partially observable states (more on that in Section 5.1.4). Inputs and outputs can be represented in a natural way by fluents as well. Requiring an input or producing an output can be expressed using the *Holds* macro, as is the case with preconditions and postconditions. For notational convenience, we define two reserved unary fluent functions, namely *HasInput* and *HasOutput*. *HasInput* denotes that the associated variable is available to the service as an input while *HasOutput* denotes that the associated value is produced as a service output.

Inputs and outputs are formalized based on the following definition:

**Definition 3.1.3** An *input formula* in  $z$  is a first-order formula  $I(z)$  with just one free state variable  $z$ , which is composed of *Holds* formulas on  $z$ , consisting exclusively of *HasInput* fluents. Equivalently, an *output formula* in  $z$  as a first-order formula  $O(z)$  with just one free state variable  $z$ , which is composed of *Holds* formulas on  $z$ , consisting exclusively of *HasOutput* fluents.

For instance, the input and output formulas for *RetrieveLocation* are, respectively,  $Holds(HasInput(request), z)$  and  $Holds(HasOutput(location), z)$ .

## 3.2 Abstract Syntax

In this section, the abstract syntax of WSSL is defined. WSSL is founded upon the fluent calculus theory that was analyzed in the previous section. Resource and data representation is inspired by the corresponding definitions in Web Service Modeling Language (WSML) [WSML Working Group 2008a], an outline of which can be found in Section 2.3.4. A complete BNF grammar for WSSL can be found in Appendix A.

### 3.2.1 Identifiers and Namespaces

Identifiers in WSSL are either IRIs or data values. An IRI can be any Unicode character sequence provided that it represents a valid and absolute IRI [Duerst and Suignard 2005]. IRIs allow any WSSL element to be linked to a concept defined in an ontology, providing the required semantics. A full IRI sequence is expected to be delimited by double quotes (""). For convenience, IRIs can be abbreviated to compact URIs, formed by a namespace prefix and a local part separated by a hash symbol (#). Also, the namespace prefix can be omitted, assuming the default namespace `http://example.org/#`. Namespaces used in a specification need to be declared at the beginning, as pairs of prefixes and full namespaces. Note that an IRI can be replaced by the symbol *nil*, if the corresponding information is not known.

WSSL *data values* can be one of the following:

- elementary data value, corresponding to the three primitive datatypes integer, decimal or string, as defined in XML Schema [Biron and Malhotra 2004]
- constructed data value, created using a *datatype wrapper*.

As in WSML, a *datatype wrapper* consists of an IRI corresponding to an XML Schema datatype (apart from the ones used in elementary data values) and a set of arguments, which can be elementary data values or *variables*. For example,

the datatype wrapper for the date type is `xsd#time(integer_hour, integer_minute, decimal_second)`. WSSL variables must start with a question mark (?) to differentiate them from other symbol sequences, such as constants.

### 3.2.2 Service Specifications

A WSSL *specification* is a 7-tuple

$$S = \langle \mathbf{service}, \mathbf{input}, \mathbf{output}, \mathbf{pre}, \mathbf{post}, \mathbf{causal}, \mathbf{default} \rangle$$

where:

- **service** is a set of identifiers offering general information about the service,
- **input** is a set of WSSL logical expressions defining *input formulas* that represent the required input of the service,
- **output** is a set of WSSL logical expressions defining *output formulas* that describe the expected output of the service,
- **pre** is a set of WSSL logical expressions defining *action precondition axioms* that detail the service preconditions, taking into account default qualifications,
- **post** is a set of WSSL logical expressions defining *state update axioms* that describe the service postconditions, including ramifications and taking into account default qualifications,
- **causal**: is a set of WSSL logical expressions defining *causal relationships* linking direct effects with their ramifications (analyzed in Section 3.2.3),
- **default**: is a default theory formalizing default qualifications for service execution (analyzed in Section 3.2.4).

Information contained in the **service** tuple can indicatively include a service name, a service operation name or information for invoking the service (see Sec-

tion 6.3.1 for more details). Note that, by default, we assume that a WSSL specification is associated with a single atomic service operation, or a black box view of a composite service. Hence, concrete services that offer more than one operations are represented by multiple WSSL specification documents. Each one of the logical expressions representing inputs, outputs, preconditions, postconditions and causal relationships is associated with an IRI that acts as an identifier.

If a service is associated with multiple state update axioms, then they are considered as alternative models of effects for the same service, similarly to using disjunctive state update axioms. Due to state updates representing a complete view of the effects of a service, outputs need to be included as positive effects, while inputs that behave like tokens (i.e. consumed during execution) are also included as negative ones. Although this may be deemed a repetition of information, the reason for additionally representing inputs and outputs separately is to be able to offer interface-only subsets, in cases where complete behavior models are not required.

An especially desirable feature in the service world is taking into account asynchronous execution, i.e. services that do not wait for a response after invocation. Asynchronous services can be easily modeled as a pair of distinct WSSL services, similarly to the invoke/receive combination of WS-BPEL [OASIS 2007], thanks to the definition of states in WSSL: the first service has no postconditions, since it simply invokes the operation, while the second has no preconditions, since they have already been checked on invocation. In this manner, the state after invoking the service is decoupled from the state after receiving the reply.

### WSSL Logical Expressions

A WSSL logical expression is defined using a first-order fragment of the fluent calculus. The alphabet consists of the following sets of symbols:

- A countable set  $\mathbf{S}$  of sorts.  $S = \{FLUENT, STATE, ACTION, SIT, ACCIDENT, BOOL\}$  with  $FLUENT < STATE$ . Sort  $BOOL$  denotes

symbols that refer to boolean values, while the rest represent the four fundamental entities of the fluent calculus, fluents, states, actions and situations, as well as the notion of accidents, required for the solution to the qualification problem (see Section 3.2.4).

- *Logical connectives:*  $\neg$  (negation),  $\wedge$  (conjunction),  $\vee$  (disjunction),  $\rightarrow$  (implication) and  $\equiv$  (equivalence).
- *Truth constants:*  $\top$  (true) and  $\perp$  (false).
- *Quantifiers:* For every sort  $s \in S$ ,  $\forall_s$  (for all),  $\exists_s$  (there exists).
- *Equality symbol:* For every sort  $s \in S$ ,  $=_s$ .
- *Auxiliary symbols:* Comma “,” and parentheses “(” ”)“.
- *Variables:* For every sort  $s \in S$ , a countably infinite set of variables  $\mathbf{V}_s$ . The family of sets  $V_s$  is denoted by  $\mathbf{V}$ .
- *Nonlogical symbols:* A set  $\mathbf{L}$  forming the WSSL signature, consisting of:
  - *Function symbols:* A countable, nonempty set  $\mathbf{FS}$  of symbols and a rank function  $r: \mathbf{FS} \rightarrow S^+ \times S$ , assigning a pair  $r(f) = (u, s)$  called *rank* to each function symbol  $f$ , with  $u$  denoting the arity ( $u > 0$ ) and  $s$  the function sort. The predefined function symbols *Do*, *State*, *HasInput* and *HasOutput* are contained at minimum. Depending on the sort of each function symbol,  $\mathbf{FS}$  contains at least three subsets:  $\mathbf{FS}_A$ , containing function symbols of the sort *ACTION*,  $\mathbf{FS}_F$ , containing function symbols of the sort *FLUENT* and  $\mathbf{FS}_{ACC}$ , containing function symbols of the sort *ACCIDENT*.
  - *Constants:* For every sort  $s \in S$ , a countable, possibly empty set of constants  $\mathbf{C}_s$ . The family of sets  $C_s$  is denoted by  $\mathbf{C}$ . In particular, the sets of state constants  $C_{STATE}$  and situation constants  $C_{SIT}$  are nonempty since the former contains at least the empty state constant  $\emptyset$  and the latter contains at least the initial situation constant  $S_0$ .

- *Predicate symbols*: A countable, nonempty set  $PS$  of symbols and a rank function  $r: PS \rightarrow S^+ \times BOOL$ , assigning a pair  $r(P) = (u, BOOL)$  called *rank* to each predicate symbol  $P$ , with  $u$  denoting the arity of the predicate. If arity is 0, then  $P$  is a propositional letter.  $PS$  contains the predefined predicate symbols *Poss*, *Causes* and *Acc*.

Note that the explicit mention of the sort in symbols such as existential and universal quantifiers can be omitted if the particular sort can be otherwise derived. All elements of sets  $FS$ ,  $PS$  are IRIs. Variables, as already mentioned, need to start with a question mark (?). It is assumed that sets  $V$ ,  $FS$ ,  $PS$  and  $C$  are disjoint. Finally, the macros *Holds* and *Ramify* (defined later on), as well as addition and subtraction of fluents from states can also be used in WSSL logical expressions.

Terms and atomic formulas are defined as follows:

1. Every constant and every variable of sort  $s$  is a term of sort  $s$ .
2. If  $t_1, \dots, t_n$  are terms, each  $t_i$  of sort  $s_i$  and  $f$  is a function symbol of sort  $s$  and arity  $s_1, \dots, s_n$ , then  $f(t_1, \dots, t_n)$  is a term of sort  $s$ .
3. Every propositional letter is an atomic formula, and so are  $\top$  and  $\perp$ .
4. If  $t_1, \dots, t_n$  are terms, each  $t_i$  of sort  $s_i$  and  $P$  is a predicate symbol of arity  $s_1, \dots, s_n$ , then  $P(t_1, \dots, t_n)$  is an atomic formula.
5. If  $t_1$  and  $t_2$  are terms of sort  $s$ , then  $t_1 =_s t_2$  is an atomic formula.

Formulas are defined as follows:

1. Every atomic formula is a formula.
2. For any two atomic formulas  $A$  and  $B$ ,  $A \wedge B$ ,  $A \vee B$ ,  $A \rightarrow B$ ,  $A \equiv B$  and  $\neg A$  are also formulas.
3. For any variable  $x_i$  of sort  $s$  and any formula  $A$ ,  $\forall_s(x_i)A$  and  $\exists_s(x_i)A$  are also formulas.

### 3.2.3 Addressing the Ramification Problem

The ramification problem poses a fundamental challenge with relation to the frame problem (and its solution in the fluent calculus): state update axioms allow only for explicit effects to be admitted as a state change, assuming everything else to be inert. Hence, accounting for ramifications (implicit, knock-on, or indirect effects) leads to the need for a modification of the format of state update axioms. As Thielscher points out (see Chapter 9 in [Thielscher 2005b]), the naive solution of treating all ramifications as direct effects leads to two major deficiencies:

- all local dependencies between effects turn into global ones, failing to capture the actual semantics of how the effects are linked
- recursive effects cannot be expressed

At the heart of the solution to the ramification problem in the fluent calculus lie causal relationships. A ramification is always linked to the direct effect (or another ramification) that brings it about; that particular relationship needs to be modeled. This is achieved by the predefined predicate *Causes* and the following definition:

**Definition 3.2.1** A *causal relationship* is a formula

$$(\forall)(\Gamma \rightarrow \text{Causes}(z, p, n, z', p', n', s))$$

where  $z, p, n, z', p'$  and  $n'$  are state variables and  $\Gamma$  is a first-order formula without any appearance of *Causes* and with  $s$  being the only situation variable contained. The semantics is the following: in situation  $s$ , under possible conditions expressed by  $\Gamma$ , the positive and negative effects  $p$  and  $n$  that have occurred cause an automatic update from state  $z$  to state  $z'$ , with positive and negative effects  $p'$  and  $n'$ .

Essentially, the *Causes* predicate is a relation between two state-effect triples with respect to a situation. Modeling of causal relationships provides the basis for inferring the ramifications of an action. In the original solution to the ramification



problem, as defined in [Thielscher 2005b], causal relationships are generalized to include the transitive closure of *Causes*, in order to express the notion of arbitrary chains of ramifications, representing causal relationships as edges in a graph and ramification inference as traversing the graph until a node with no outgoing edges is reached (a so-called sink). However, in the case of services and WSSL, specification requirements are much simpler: since causal relationships are not always expected to be expressed by providers, some relationships can be derived given a set of services that participate in a service composition, in order to determine the consistency of the composition. Hence, there is no need to express arbitrary chains, since the derivation will, at best, yield concrete direct causal relationships between condition pairs.

Taking the above into account, inferring ramifications is expressed by a simplified version of the macro *Ramify* (compared to its original form in [Thielscher 2005b]), defined as follows:

$$Ramify(z, p, n, z', s) \stackrel{def}{=} (\exists p', n')(Causes(z - n + p, p, n, z', p', n', s)).$$

The semantics is that in situation  $s$ , state  $z$  gets updated by positive effects  $p$  and negative effects  $n$  as well as the application of the causal relationship expressed by *Causes*, leading to the final state  $z'$ . The final step for solving the ramification problem is integrating ramifications into state update axioms, so that they offer a complete view of what is caused after the execution of an action. The state update axioms with ramifications are defined as follows:

$$Poss(A(x), s) \rightarrow (\exists y)(\Delta(z) \wedge Ramify(z, \theta^+, \theta^-, z', Do(A(x), s)))$$

with  $\Delta(z)$  a state formula with free variables among  $x, y, z$  and  $\theta^+, \theta^-$  finite states with variables among  $x, y$ . The semantics is slightly modified as follows: provided that an action  $A$  is possible at a situation  $s$ , then the action execution at situation  $s$  results in a successor state  $z'$  which is the result of positive effects  $\theta^+$  and negative effects  $\theta^-$  occurring at state  $z$ , in turn causing ramifications that lead to  $z'$ , under possible additional conditions expressed by formula  $\Delta(z)$  (again, in the simplest case,  $\Delta(z) \equiv \top$ ).

To illustrate how the solution to the ramification problem can be applied to the running example, let us consider again the case of the *RetrieveLocation* task. A successful execution means that the vehicle status has been received; in turn, this means that a mechanic log is generated, based on the gathered information. This causal relationship is expressed as follows:

$$\begin{aligned} ?p = & \text{HasOutput}(\text{status}) + \text{Retrieved}(\text{status}, \text{vehicle}) \wedge \\ ?n = & \text{HasInput}(\text{request}) \Rightarrow \text{Causes}(?z, ?p, ?n, ?z + \\ & \text{Generated}(\text{mechlog}), ?p + \text{Generated}(\text{mechlog}), ?n, ?s) \end{aligned}$$

In order for this causal relationship to be taken into account when determining the effects of executing *RetrieveLocation*, the state update axiom associated with the task must be slightly modified based on the *Ramify* macro definition:

$$\begin{aligned} \text{Poss}(\text{RetrieveLocation}, s) \Rightarrow & (\exists \text{location}, \text{request}) \\ \text{Ramify}(z, \text{Retrieved}(\text{location}, \text{user}), & \text{HasInput}(\text{request}), z', \\ & \text{Do}(\text{RetrieveLocation}, s)) \end{aligned}$$

### 3.2.4 Addressing the Qualification Problem

While the ramification problem deals with presenting a complete account of the effects of an action, the qualification problem tackles the dual issue of accounting for all possible preconditions that are necessary for an action. This involves taking into account expectations that are otherwise assumed to be always satisfied, but which can explain unsuccessful executions when all normal preconditions hold. This is realized through the inclusion in the fluent calculus of sort *ACCIDENT*, along with  $FS_{ACC}$ , a finite set of function symbols of that sort and an additional predefined predicate, *Acc*. *Acc* is of arity 2, taking as input one argument of sort *ACCIDENT* and one of sort *SIT*.  $Acc(c, s)$  carries the meaning that accident *c* happened in situation *s*.

In order to assume away accidents (we assume they do not happen except if we cannot do otherwise), default logic formalisms [Reiter 1980] must be employed.

A default theory is a pair  $\langle \Delta, O \rangle$  where  $O$  is a possibly empty set of *observations*, situation formulas that are known to be true and  $\Delta$  is a set of default rules of the form:  $\frac{\text{Prerequisite:Justification}(s)}{\text{Conclusion}}$ . [Thielscher 2005b] proposes the following default theory:

$$\Delta = (\{ \frac{\neg Acc(c,s)}{\neg Acc(c,s)} \}, \Sigma \cup O)$$

where  $O$  is a set of observations, situation formulas that are true and  $\Sigma$  is the domain axiomatization. The rule is essentially a single universal default on the non-occurrence of all accidents, an application of the closed world assumption on accidents happening. As long as the observations are in line with the expected effects of an action, then no accident needs to be considered as having taken place.

Accidents are then integrated into state update and precondition axioms. To express the default case of the effects of an action where no accident has taken place, we just include the conjunct  $(\forall c)\neg Acc(c, s)$  in the state update axiom. Equivalently, precondition axioms are rewritten in the following form:

$$Poss(A(x), s) \equiv [(\forall c)\neg Acc(c, s) \rightarrow \Pi_A(x, s)]$$

meaning that an action is possible at a situation provided that no accidents have happened and the preconditions are true.

Apart from these default modifications, where no accident has happened, one can think of other clauses to include in a state update or precondition axiom. For instance, we may want to express what effects are brought upon by a particular accident happening. This can be accomplished by adding a disjunct to the state update axiom of the action that expresses that connection. Similarly, if we want to express new preconditions for an action to account for an accident happening then a conjunct can be added containing an implication with an accident clause on one side and the associated preconditions on the other. However, such detailed modeling of accidents may not be suitable for the service case, since one can expect a service designer or provider to provide minimum details concerning accidents, such as accidental effects, but not further information concerning conditions for these accidents happening.

An application of the solution to the qualification problem in the running example involves the postconditions of the *EReport* task. Under normal circumstances, a successful execution leads to a report being delivered via e-mail to the user. However, in case no report is delivered, even though all preconditions were known to hold before execution, then this can be explained by the occurrence of a delivery failure. This behavior can be expressed in the right-hand side of the action precondition axiom as follows:

$$\begin{aligned}
& (\forall ?c) \neg Acc(?c, ?s) \wedge (State(Do(EReport, ?s)) = State(?s) + \\
& HasOutput(report) + Emailed(report) - HasInput(invoice)) \vee (\exists ?deliv) \\
& (Acc(Failure(?deliv, ?s)) \wedge State(Do(EReport, ?s)) = State(?s))
\end{aligned}$$

### 3.2.5 WSSL specification of the running example

Having defined the complete abstract syntax for WSSL, we turn our focus to the running example, defined in Section 2.1, in order to express WSSL specifications for the service tasks that are described in it. Tables 3.1 and 3.2 offer indicative specifications for these tasks. To achieve a more condensed representation, the following simplifications are adopted:

- *Poss* predicates are omitted; only the right-hand side of action precondition axioms is included.
- The left-hand side of state update axioms, as well as the existential quantification at the beginning of the right-hand side are omitted.
- No-accident clauses of the form  $(\forall ?c) \neg Acc(?c, ?s)$  are included only in the case where there is also an accident clause.
- In each state update axiom, existential quantification is implied. Additionally,  $?z\_in \equiv State(?s\_in)$  and  $?z\_out \equiv State(Do(A(?x), ?s\_in))$ .

Table 3.1: WSSL Specifications for the running example - part 1

<b>Service</b>	<b>Inputs</b>
ReceiveSMS/Call	$Holds(HasInput(sms), ?z\_in) / Holds(HasInput(call), ?z\_in)$
RetrieveLocation	$Holds(HasInput(request), ?z\_in)$
RetrieveDiag	$Holds(HasInput(request), ?z\_in)$
FindMech	$Holds(HasInput(status), ?z\_in) \wedge Holds(HasInput(location), ?z\_in)$
ReceivePay	$Holds(HasInput(payform), ?z\_in)$
EReport	$Holds(HasInput(invoice), ?z\_in)$
MReport	$Holds(HasInput(invoice), ?z\_in)$
<b>Service</b>	<b>Preconditions</b>
ReceiveSMS/Call	$Holds(CallCenterUp, ?z\_in)$
RetrieveLocation	$Holds(GPSActive(user), ?z\_in)$
RetrieveDiag	$Holds(System.Active(vehicle), ?z\_in)$
FindMech	$Holds(Retrieved(location, user), ?z\_in) \wedge$ $Holds(Retrieved(status, vehicle), ?z\_in) \wedge \neg Holds(Solved(status, location), ?z\_in)$
ReceivePay	$Holds(Solved(status, location), ?z\_in) \wedge \neg Holds(Deactivated(creditCard), ?z\_in)$
EReport	$Holds(PayCompleted(payform), ?z\_in) \wedge$ $Holds(Generated(mechlog), ?z\_in) \wedge \neg Holds(Emailed(report), ?z\_in)$
MReport	$Holds(PayCompleted(payform), ?z\_in) \wedge$ $Holds(Generated(mechlog), ?z\_in) \wedge \neg Holds(Delivered(report), ?z\_in)$

Table 3.2: WSSL Specifications for the running example - part 2

Service	Outputs and Postconditions
ReceiveSMS	$?z\_out = ?z\_in + HasOutput(request) + Received(request, sms) - HasInput(sms)$
ReceiveCall	$?z\_out = ?z\_in + HasOutput(request) + Received(request, call) - HasInput(call)$
RetrieveLocation	$?z\_out = ?z\_in + HasOutput(location) + Retrieved(location, user) - HasInput(request)$
RetrieveDiag	$RamiIy(?z\_in, HasOutput(status) + Retrieved(status, vehicle), HasInput(request), ?z\_out)$
FindMech	$?z\_out = ?z\_in + HasOutput(payform) + Solved(status, location) - HasInput(status) - HasInput(location)$
ReceivePay	$RamiIy(?z\_in, HasOutput(invoice) + PayCompleted(payform), HasInput(payform), ?z\_out)$
EReport	$(\forall ?c) \neg Acc(?c, ?s\_in)(?z\_out = ?z\_in + HasOutput(report) + Emailed(report) - HasInput(invoice)) \vee (\exists ?deliv)(Acc(Failure(?deliv), ?s\_in) \wedge ?z\_out = ?z\_in)$
MReport	$?z\_out = ?z\_in + HasOutput(report) + Delivered(report) - HasInput(invoice)$
<b>Causal Relationships</b>	
	$?p = HasOutput(status) + Retrieved(status, vehicle) \wedge ?n = HasInput(request) \Rightarrow Causes(?z, ?p, ?n, ?z + Generated(mechlog), ?p + Generated(mechlog), ?n, ?s)$
	$DailyLimitReached(payform, creditCard) \wedge ?p = HasOutput(invoice) + PayCompleted(payform) \wedge ?n = HasInput(payform) \Rightarrow Causes(?z, ?p, ?n, ?z + Deactivated(creditCard), ?p + Deactivated(creditCard), ?n, ?s)$

### 3.3 Surface Syntax

The abstract syntax defined in the previous section is not easily interpreted by humans. In order to increase human readability, we define a surface syntax. Similarly to WSML [WSML Working Group 2008b], we define a mapping function  $tr$  that takes as input a WSSL specification written using the abstract syntax and returns the equivalent specification using the surface syntax. The application of the mapping function to all elements of a WSSL specification follows. Underlined words represent reserved keywords for the WSSL surface syntax, while bold words represents WSSL specification parts. We assume that an indicative subset of what can be included as service-related information in the service tuple, namely service name and service grounding.

$$\begin{aligned}
 tr(\langle name, \mathbf{input}, \mathbf{output}, \mathbf{pre}, \mathbf{post}, \mathbf{causal}, \mathbf{default} \rangle) = & \\
 & \underline{service} \ tr(name) \ tr(grounding) \\
 & \quad \underline{input} \ tr(\mathbf{input}) \\
 & \quad \underline{output} \ tr(\mathbf{output}) \\
 & \quad \underline{precondition} \ tr(\mathbf{pre}) \\
 & \quad \underline{postcondition} \ tr(\mathbf{post}) \\
 & \quad \underline{causalrelation} \ tr(\mathbf{causal}) \\
 & \quad \underline{defaulttheory} \ tr(\mathbf{default})
 \end{aligned}$$

IRIs and data values are unaffected by the translation process, staying the same in both abstract and surface syntaxes. Namespace declarations are preceded by a namespace keyword, followed by a list in braces, separated by commas. Table 3.3 details the mapping process from abstract to surface syntax for WSSL logical expressions, through a translation function  $tr()$ . The equality symbol, as well as predicate and function symbols and variable names stay the same in the surface syntax. Observations in a default theory are translated in the same way as

Abstract Syntax	Surface Syntax
$tr(\neg\phi)$	<i>not</i> $tr(\phi)$
$tr(\phi \wedge \psi)$	$tr(\phi)$ <i>and</i> $tr(\psi)$
$tr(\phi \vee \psi)$	$tr(\phi)$ <i>or</i> $tr(\psi)$
$tr(\phi \rightarrow \psi)$	$tr(\phi)$ <i>implies</i> $tr(\psi)$
$tr(\phi \equiv \psi)$	$tr(\phi)$ <i>equivalent</i> $tr(\psi)$
$tr(\top)$	<i>true</i>
$tr(\perp)$	<i>false</i>
$tr(\forall?x(\phi))$	<i>forall</i> $?x$ ( $tr(\phi)$ )
$tr(\exists?x(\phi))$	<i>exists</i> $?x$ ( $tr(\phi)$ )

Table 3.3: Surface syntax for WSSL logical expressions

WSSL logical expressions. Default rules are translated as follows:

$$tr\left(\frac{\textit{Prerequisite} : \textit{Justification}(s)}{\textit{Conclusion}}\right) =$$

$$\textit{if } tr(\textit{Prerequisite}) \textit{ assuming } tr(\textit{Justification}(s)) \textit{ then } tr(\textit{Conclusion})$$

Prerequisites, justifications and conclusions are translated as WSSL logical expressions.

### 3.4 Semantics

The semantics for WSSL is defined based on the standard model theory for classical first-order logic [Galton 1990], augmented by the semantics for IRIs as defined in WSML [WSML Working Group 2008b]. Also, the associated default theory follows the semantics first presented in [Reiter 1980]. The main additional aspect brought on by the need to interpret IRIs is the notion of abstract and concrete domains. An abstract domain gives us flexibility in interpreting IRIs as any kind of abstract object, while a concrete domain allows for the interpretation of elementary data values. A concrete domain needs to support all three elementary data types used by WSSL (integer, decimal and string). For instance, it can



be equal to the union of the sets of integer numbers, finite-length sequences of decimal digits (preceded or not by the minus symbol) and finite-length sequences of Unicode characters.

A WSSL interpretation is a 6-tuple  $\mathcal{I} = \langle U, D, \mathcal{I}_C, \mathcal{I}_F, \mathcal{I}_P, B \rangle$  where:

- $U$  is the abstract domain of interpretation, a non-empty countable set used to interpret IRIs. Note that the symbol *nil*, which may substitute an IRI, needs to be replaced by a unique IRI before interpretation.
- $D$  is the concrete domain of interpretation, a non-empty set disjoint from  $U$ , used to interpret elementary data values.
- $\mathcal{I}_C$  is a mapping from individual constants to elements of  $U$  and  $D$ .
- $\mathcal{I}_F$  is a mapping from function symbols to functions over  $U$  and  $D$ .
- $\mathcal{I}_P$  is a mapping from predicate symbols to predicates over  $U$  and  $D$ .
- $B$  is an assignment from a variable to an element of  $U \cup D$

The interpretation of constants depends on whether the constant is an IRI or an elementary data value. In the former case,  $\mathcal{I}_C(c) = u \in U$ , while in the latter case  $\mathcal{I}_C(c) = d \in D$ . A similar distinction applies to function symbols. If the function symbol represents a data wrapper function (that creates a constructed data value), it is interpreted as a function over the concrete domain:  $\mathcal{I}_F(f)^i = D^i \rightarrow D$ , with  $i$  denoting the arity. Otherwise, we work on the abstract domain and  $\mathcal{I}_F(f)^i = U^i \rightarrow U$ . Predicate symbols are interpreted as a subset of both domains:  $\mathcal{I}_P(p) \subseteq D \cup U$ .

In order to define interpretation of terms, we first need to handle variables. For each variable, there can be a number of variable assignments  $B$ , assigning each variable  $v$  to an individual  $v^B \in U \cup D$ . A variable assignment in the concrete domain  $v^B \in D$  is called a concrete variable while a variable assignment in the union of the domains  $v^B \in U \cup D$  is called an abstract variable. Given that definition, terms are interpreted as follows:

- If a term is a variable with assignment  $B$ , then  $t^{\mathcal{I},B} = t^B$
- If a term is a function  $f(t_1, \dots, t_n)$ , then  $t^{\mathcal{I},B} = \mathcal{I}_{\mathcal{F}}(f)(t_1^{\mathcal{I},B}, \dots, t_n^{\mathcal{I},B})$

### 3.4.1 Satisfaction and Entailment

Having interpreted terms, we now move on to formula interpretation. We define the notion of satisfaction for a WSSL formula, given an interpretation  $\mathcal{I}$  and under a variable assignment  $B$  (e.g., for formulas with free variables). If a formula  $\phi$  is satisfied by an interpretation  $\mathcal{I}$  under a variable assignment  $B$ , we write  $(\mathcal{I}, B) \models \phi$ . If  $\phi$  is satisfied by  $\mathcal{I}$  under any possible variable assignment, we write  $\mathcal{I} \models \phi$ , which can also be read as  $\mathcal{I}$  is a model of  $\phi$  or  $\phi$  is true in  $\mathcal{I}$ . The opposite ( $\mathcal{I}$  is not a model of  $\phi$ ) is denoted by  $\mathcal{I} \not\models \phi$ . The satisfaction relation, for a formula  $\phi$  under variable assignment  $B$  is defined as follows:

- $(\mathcal{I}, B) \models \top$  (and equivalently  $\mathcal{I} \models \top$ )
- $(\mathcal{I}, B) \not\models \perp$  (and equivalently  $\mathcal{I} \not\models \perp$ )
- $(\mathcal{I}, B) \models t_1 = t_2$  iff  $t_1^{\mathcal{I},B} = t_2^{\mathcal{I},B}$
- $(\mathcal{I}, B) \models p(t_1, \dots, t_n)$  iff  $(t_1^{\mathcal{I},B}, \dots, t_n^{\mathcal{I},B}) \in \mathcal{I}_{\mathcal{P}}(p)$
- $(\mathcal{I}, B) \models \neg\phi$  iff  $(\mathcal{I}, B) \not\models \phi$

For any formulas  $\phi_1, \phi_2$ :

- $(\mathcal{I}, B) \models \phi_1 \wedge \phi_2$  iff  $(\mathcal{I}, B) \models \phi_1$  and  $(\mathcal{I}, B) \models \phi_2$
- $(\mathcal{I}, B) \models \phi_1 \vee \phi_2$  iff  $(\mathcal{I}, B) \models \phi_1$  or  $(\mathcal{I}, B) \models \phi_2$
- $(\mathcal{I}, B) \models \phi_1 \rightarrow \phi_2$  iff  $(\mathcal{I}, B) \not\models \phi_1$  or  $(\mathcal{I}, B) \models \phi_2$
- $(\mathcal{I}, B) \models \phi_1 \equiv \phi_2$  iff  $(\mathcal{I}, B) \models \phi_1 \rightarrow \phi_2$  and  $(\mathcal{I}, B) \models \phi_2 \rightarrow \phi_1$

Given a concrete variable  $x$  ( $x^B \in D$ ) or an abstract variable  $x$ , ( $x^B \in D \cup U$ )  
 $(\mathcal{I}, B) \models \forall x \cdot \phi(x)$  iff  $\mathcal{I} \models \phi$  and  $(\mathcal{I}, B) \models \exists x \cdot \phi(x)$  iff for some  $B$ ,  $(\mathcal{I}, B) \models \phi$

Other definitions from classical logic apply:

- A formula  $\phi$  is satisfiable if there is an interpretation  $\mathcal{I}$  that is a model of  $\phi$  ( $\mathcal{I} \models \phi$ ).
- A formula  $\phi$  is valid if every possible interpretation  $\mathcal{I}$  is a model of  $\phi$ .
- An interpretation  $\mathcal{I}$  is a model of a theory  $\Phi$  if for every  $\phi \in \Phi$ ,  $\mathcal{I} \models \phi$ .
- A theory  $\Phi$  is satisfiable iff it has a model.

For two schemas  $A$  and  $B$ ,  $A$  entails  $B$ , denoted by  $A \models B$ , iff the set of all models of  $A$  is a subset of the set of all models of  $B$ , written as  $M(A) \subset M(B)$ .  $A$  and  $B$  are equivalent, denoted by  $A \equiv B$  iff  $M(A) = M(B)$ .

### 3.5 WSSL/XML

The abstract and surface syntaxes presented in the preceding sections are intended for human consumption. In order to provide machine readability and facilitate exchange of WSSL documents on the Web, an XML syntax is proposed in this section, named WSSL/XML. As a basic namespace for all elements in this syntax, we use `http://www.example.org/wssl#`. A WSSL/XML specification is enclosed in a `wssl` tag of the following form:

```
<wssl xmlns="http://www.example.org/wssl#"
      variant="http://www.example.org/wssl#WSSL">
</wssl>
```

Listing 3.1: WSSL/XML root element

The default assumed value for the `variant` attribute is "WSSL". If the document contains a simplified WSSL specification, where, for instance, ramifications and qualifications are not supported, then the `variant` attribute should be set differently. General information about the service (service name and invocation) are enclosed in a `service` tag.

Each one of the inputs contained in the specification is translated to an `input` tag, with a `name` attribute containing the name (in the form of an IRI of the input).

Similarly, outputs correspond to *output* tags, preconditions correspond to *precondition* tags, postconditions correspond to *postcondition* tags and default theories correspond to *default* tags. Full IRIs remain as they are in the XML syntax, while abbreviated IRIs need to be expanded. Note that any other occurrence of inputs or outputs in the form of HasInput or HasOutput fluents is omitted for the sake of readability.

The left-hand side of preconditions (containing *poss*) is omitted; the XML code that corresponds to the logical expression of the right-hand side is placed directly under the *precondition* tag. Similarly, only the right-hand side of postconditions is included. Also, instead of directly translating  $State(Do(A(?x), ?s\_in))$  and  $State(?s\_in)$ , we replace them with two simple state terms,  $z_{out}$  and  $z_{in}$ , respectively. Finally, the conjunct  $(\forall c)\neg Acc(c, s)$  for non-accident cases is optional.

Data values are translated to *term* tags containing their type as an attribute. If the data value is produced by a datatype wrapper, then an inner *argument* tag is added for each one of the arguments. Arbitrary functions are translated in the same way, with the type attribute corresponding to the function name.

A WSSL default theory is represented by *observation* and *rule* tags, corresponding to the observations and rules that constitute the theory, respectively. Following the format of a default rule, a *rule* tag contains one optional *prerequisite* tag, one *conclusion* tag and one or more *prerequisite* tags. Finally, causal relationships are represented by *causal* tags.

Translation of WSSL logical expressions is based on the XML syntaxes for WSML and RuleML [Boley et al. 2012] and is analyzed in Table 3.4.  $expr_i$  stands for any arbitrary logical expression, while  $tr()$  represents again the translation function. Simple terms are translated to *term* tags with a *type* attribute denoting their type (xs:integer, xs:decimal, xs:string, or variable). If the term type is a variable, then the term tag contains a *var* tag. Predefined functions, such as *Do* and *State* are translated as any arbitrary function. A complete XML schema for WSSL/XML, as well as some example WSSL/XML specifications can be found in Appendix B.

	Abstract Syntax	XML Syntax
True	$\top$	<code>&lt;true/&gt;</code>
False	$\perp$	<code>&lt;false/&gt;</code>
Empty State	$\emptyset$	<code>&lt;empty/&gt;</code>
Negation	$tr(\neg expr_1)$	<code>&lt;neg&gt; tr(expr<sub>1</sub>) &lt;/neg&gt;</code>
Conjunction	$tr(expr_1 \wedge expr_2)$	<code>&lt;and&gt; tr(expr<sub>1</sub>) tr(expr<sub>2</sub>) &lt;/and&gt;</code>
Disjunction	$tr(expr_1 \vee expr_2)$	<code>&lt;or&gt; tr(expr<sub>1</sub>) tr(expr<sub>2</sub>) &lt;/or&gt;</code>
Implication	$tr(expr_1 \rightarrow expr_2)$	<code>&lt;implies&gt; tr(expr<sub>1</sub>) tr(expr<sub>2</sub>) &lt;/implies&gt;</code>
Equivalence	$tr(expr_1 \equiv expr_2)$	<code>&lt;equivalent&gt; tr(expr<sub>1</sub>) tr(expr<sub>2</sub>) &lt;/equivalent&gt;</code>
Variable	$tr(?varname)$	<code>&lt;var name="varname"/&gt;</code>
Universal Quantification	$tr(\forall(?varname) \cdot expr_1)$	<code>&lt;forall&gt; &lt;var name="varname"/&gt; tr(expr<sub>1</sub>) &lt;/forall&gt;</code>
Existential Quantification	$tr(\exists(?varname) \cdot expr_1)$	<code>&lt;exists&gt; &lt;var name="varname"/&gt; tr(expr<sub>1</sub>) &lt;/exists&gt;</code>
Function	$tr(f(term_1, \dots, term_n))$	<code>&lt;term name="func"&gt; tr(term<sub>1</sub>)...tr(term<sub>n</sub>)&lt;/term&gt;</code>
Equality	$tr(term_1 = term_2)$	<code>&lt;equal&gt; tr(term<sub>1</sub>) tr(term<sub>2</sub>)&lt;/equal&gt;</code>
Predicate	$tr(p(term_1, \dots, term_n))$	<code>&lt;predicate name="pred"&gt; tr(term<sub>1</sub>)...tr(term<sub>n</sub>)&lt;/predicate&gt;</code>
Holds	$tr(Holds(f, z))$	<code>&lt;holds state="z"&gt; tr(f) &lt;/holds&gt;</code>
Ramify	$tr(Ramify(z, p, n, z'))$	<code>&lt;ramify&gt; tr(z)tr(p)tr(n)tr(z') &lt;/ramify&gt;</code>
Minus	$tr(z - f)$	<code>&lt;minus&gt; tr(z)tr(f) &lt;/minus&gt;</code>
Plus	$tr(z + f)$	<code>&lt;plus&gt; tr(z)tr(f) &lt;/plus&gt;</code>
Composition	$tr(f_1 \circ \dots \circ f_n)$	<code>&lt;circ&gt; tr(f<sub>1</sub>)...tr(f<sub>n</sub>) &lt;/circ&gt;</code>

Table 3.4: XML syntax for WSSL logical expressions



# Chapter 4

## WSSL Extensions

### Contents

---

<b>4.1</b>	<b>Composition</b>	<b>98</b>
4.1.1	Calculating composite preconditions and postconditions	98
4.1.2	WSSL for Composition	103
<b>4.2</b>	<b>Quality of Service</b>	<b>105</b>
4.2.1	WSSL QoS Profiles	106
4.2.2	Correctness of QoS Profiles	108
4.2.3	Local QoS Goal Matching	109
4.2.4	QoS Metrics Analysis and Aggregation	110
<b>4.3</b>	<b>Partial Observability</b>	<b>113</b>
4.3.1	Incomplete States	115
4.3.2	Knowledge States	116

---

The present chapter proposes a series of extensions to the initial definition of WSSL that was presented in the previous chapter. The extensions target three interesting directions in service science. The first extension covers specification of composite services and enables WSSL to be employed as a service composition language as well. The second extension covered in this chapter imbues WSSL with the

ability to express QoS profiles, aiming to achieve QoS-awareness in service specification. Finally, a way to handle partial observability in service specifications is proposed, by extending WSSL with incomplete state and knowledge modeling.

## 4.1 Composition

The definition of WSSL in Chapter 3 allows for black-box specifications of services where only IOPEs are considered, disregarding any knowledge about its control and data flow. In order to be able to employ WSSL for composition, we need to extend the language definition to include the modeling of fundamental control constructs. Composite services can then be modeled as complex actions using one or more of these constructs, depending on the control flow of the composition. To achieve that, we first need to express preconditions and postconditions of fundamental compositions that use a single control construct.

### 4.1.1 Calculating composite preconditions and postconditions

The following control constructs are examined: sequence, AND-Split/AND-Join, AND-Split/DISC-Join, OR-Split/OR-Join, OR-Split/DISC-Join, XOR-Split/XOR-Join, conditional execution and loops. DISC-Join is a *m*-out-of-*n* join, also called a *discriminator*. These constructs correspond to the list of abstract composition patterns that is defined in [Jaeger et al. 2004] based on the list of workflow patterns in [van der Aalst et al. 2003], since after analyzing the updated list in [Russell et al. 2006], no additional relevant composition patterns could be yielded. Hence, this set of patterns (and the equivalent set of control constructs) is considered complete.

The results that follow are based on composite specification derivation, as defined in [Baryannis et al. 2012]. In all cases, and without loss of generality, we consider compositions of two services represented by actions  $a_1$  and  $a_2$ , with



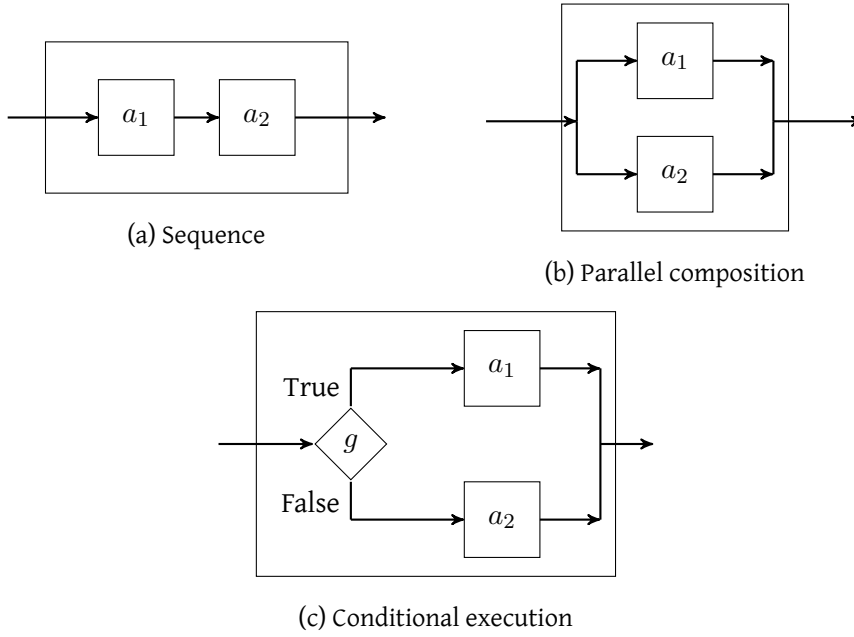


Figure 4.1: Fundamental control constructs for service composition

state update axioms expressed in the following simplified form:

$$\begin{aligned}
 Poss(a_1, s_1) &\rightarrow (State(Do(a_1, s_1)) = State(s_1) + \theta_1^+ - \theta_1^-) \\
 Poss(a_2, s_2) &\rightarrow (State(Do(a_2, s_2)) = State(s_2) + \theta_2^+ - \theta_2^-) \quad (4.1)
 \end{aligned}$$

### Sequence

In the case of sequential execution, denoted by  $a_1; a_2$  and following the pattern shown in Fig. 4.1a, the situation in which service  $a_2$  is executed is the same as the one that results after executing service  $a_1$ ; this fact is expressed formally as  $s_2 = Do(a_1, s_1)$ . Based on this, as well as the fact that we want both state update axioms of Axiom 4.1 to hold, we result in Axiom 4.2:

$$\begin{aligned}
 &Poss(a_1, s_1) \wedge Poss(a_2, Do(a_1, s_1)) \rightarrow \\
 &State(Do(a_2, Do(a_1, s_1))) = State(s_1) + \theta_1^+ - \theta_1^- + \theta_2^+ - \theta_2^- \quad (4.2)
 \end{aligned}$$

### AND-Split/AND-Join

Parallel composition, shown in Fig. 4.1b comes in many different variations. AND-Split/AND-Join, denoted by  $a_1 \cdot a_2$ , is one of them; there are two diverging branches of activities that are executed concurrently, which eventually converge after activities in both branches have completed successfully. In this case, the situation in which both services are executed is the same ( $s_1 \equiv s_2 \equiv s$ ) and the situation afterwards is again the same ( $Do(a_1, s_1) \equiv Do(a_2, s_2) \equiv Do(a_*, s)$ ), since both services need to complete execution for the composition to be considered complete. This knowledge leads to Axiom 4.3:

$$\begin{aligned} & Poss(a_1, s) \wedge Poss(a_2, s) \rightarrow \\ & State(Do(a_1 \cdot a_2, s)) = State(s) + \theta_1^+ - \theta_1^- + \theta_2^+ - \theta_2^- \end{aligned} \quad (4.3)$$

### OR-Split/OR-Join, AND-Split/DISC-Join, OR-Split/DISC-Join

OR-Split/OR-Join, denoted by  $a_1 + a_2$ , differs from AND-Split/AND-Join in that not all of the diverging branches are necessarily activated. Instead, a mechanism selects one or more of them to be executed each time. Also, at the merging stage there is no need for synchronization between the converging branches. This behavior is reflected in Axiom 4.4:

$$\begin{aligned} & Poss(a_1, s) \wedge Poss(a_2, s) \rightarrow \\ & (State(Do(a_1 + a_2, s)) = State(s) + \theta_1^+ - \theta_1^-) \vee \\ & (State(Do(a_1 + a_2, s)) = State(s) + \theta_2^+ - \theta_2^-) \vee \\ & (State(Do(a_1 + a_2, s)) = State(s) + \theta_1^+ - \theta_1^- + \theta_2^+ - \theta_2^-) \end{aligned} \quad (4.4)$$

The behavior of AND-Split/DISC-Join and OR-Split/DISC-Join is also expressed with Axiom 4.4, since the state update of a DISC-Join also translates to either of the two, or both services completing execution. Hence, the remainder of this section does not refer to them again, keeping only OR-Split/OR-Join as a representative of such behavior.

In order to make sure that the solution to the frame problem also holds for all parallel compositions defined so far, we need to assume that  $(\theta_1^+ + \theta_2^+)$  and  $(\theta_1^- + \theta_2^-)$  are disjoint. This practically means that no effect of one service is negated by the other. This does not go against the nature of AND/OR parallel compositions, where it is almost always expected to execute independent services in parallel so as to reduce execution time. The only case where a parallel execution may involve services that are related to each other is when we want to model racing conditions; this case is more accurately modeled by the XOR-Split/XOR-Join case, examined next.

### XOR-Split/XOR-Join

The final variation of parallel composition is XOR-Split/XOR-Join, denoted by  $a_1 \oplus a_2$ . In this case, only one of the diverging branches is allowed to be executed and is expected to provide results at the end. Note that this behavior is quite similar to conditional execution (defined below), with one defining difference: we have no knowledge of the condition that causes one of the two services to be selected; we just expect only one of them to provide results. In that sense, XOR-Split/XOR-Join can also model behaviors of a racing nature, where the first service to provide results is the one that is actually considered part of the composite process. Axiom 4.5 encodes the behavior of XOR-Split/XOR-Join:

$$\begin{aligned}
 & Poss(a_1, s) \wedge Poss(a_2, s) \rightarrow \\
 & (State(Do(a_1 \oplus a_2, s)) = State(s) + \theta_1^+ - \theta_1^-) \oplus \\
 & (State(Do(a_1 \oplus a_2, s)) = State(s) + \theta_2^+ - \theta_2^-) \quad (4.5)
 \end{aligned}$$

Note that regardless of the particular type of parallel composition (AND, OR, XOR), both preconditions of the services executed in parallel are required to be true. This condition may appear too strong for OR and XOR cases, but it stems directly from the fact that, at design time, we do not know which branch is going to be executed; hence, we cannot disregard either precondition. If such knowledge is available at runtime, then preconditions may be adapted accordingly.

### Conditional Execution

Conditional constructs, such as if-then-else or switch statements, evaluate a condition in order to decide which branch will be executed. Similarly to the XOR-Split/XOR-Join pattern, only one of the branches is selected, based on the truth value of the condition. In an if-then-else composition of the form *IF g THEN a<sub>1</sub> ELSE a<sub>2</sub>*, denoted by  $If(g, a_1, a_2)$  and illustrated in Fig. 4.1c, if condition  $g$  is true,  $a_1$  is executed; if it is false,  $a_2$  is executed. The condition essentially determines not only which precondition is to be checked, but also which postcondition will be applied after a successful execution. This is expressed in Axiom 4.6:

$$\begin{aligned}
& [(Holds(f, s) \wedge Poss(a_1, s)) \vee (\neg Holds(f, s) \wedge Poss(a_2, s))] \rightarrow \\
& [(Holds(f, s) \wedge (State(Do(If(f, a_1, a_2), s)) = State(s) + \theta_1^+ - \theta_1^-)) \vee \\
& (\neg Holds(f, s) \wedge (State(Do(If(f, a_1, a_2), s)) = State(s) + \theta_2^+ - \theta_2^-))] \quad (4.6)
\end{aligned}$$

### Loops

Loops allow for the repeated execution of a task or a process until a condition (the loop guard) ceases to hold. This poses a significant challenge as there is no *a priori* knowledge of how many iterations will be performed, which leads to precondition and postcondition expressions that are interminable. They can only be made finite if the number of iterations is known or limited beforehand. Attempting to encode such a behavior results in the incomplete Axiom 4.7 (where  $g$  is the loop guard and a looped action is denoted as  $Loop(g, a_1)$ ):

$$\begin{aligned}
& (Holds(g, s) \rightarrow Poss(a_1, s)) \wedge \\
& (Holds(g, Do(a_1, s)) \rightarrow Poss(a_1, Do(a_1, s))) \wedge \dots \rightarrow \\
& \neg Holds(g, s) \wedge State(Do(Loop(g, a_1))) = State(s) \wedge \\
& Holds(g, s) \wedge \neg Holds(g, Do(a_1, s)) \wedge \\
& State(Do(Loop(g, a_1))) = State(s) + \theta_1^+ - \theta_1^- \dots \quad (4.7)
\end{aligned}$$

### 4.1.2 WSSL for Composition

Based on the calculated state update axioms for the different control constructs, we present a series of definitions that constitute the extension of WSSL in the direction of supporting composite service specification and service composition. The first definition extends the WSSL signature in order to include control constructs:

**Definition 4.1.1** A tuple  $S \cup S_C$  is an extended WSSL signature for composition if  $S$  is a WSSL signature (as defined in Section 3.2) and  $S_C$  is a set of function symbols, defined as follows:

- $\epsilon : ACTION$  (empty action)
- $If: FLUENT \times ACTION \times ACTION \rightarrow ACTION$  (conditional execution)
- $Loop: FLUENT \times ACTION \rightarrow ACTION$  (iterative execution)
- $;, \cdot, +, \oplus : ACTION \times ACTION \rightarrow ACTION$  (sequence, AND-Split/AND-Join, OR-Split/OR-Join and XOR-Split/XOR-Join, respectively)

It follows that the foundational axioms that govern the fluent calculus and WSSL, as expressed in Chapter 3, need to be extended in order to account for the newly introduced function symbols; this is achieved by Definitions 4.1.2 and 4.1.3. The extension is based on the analysis so far in this section, as well as the methodology followed by [Thielscher 2001a] in the attempt to introduce concurrency in the fluent calculus.

**Definition 4.1.2** The foundational axioms for **preconditions** consist of:

1.  $Poss(\epsilon, s) \equiv T$
2.  $Poss(a_1; a_2, s) \equiv Poss(a_1, s) \wedge Poss(a_2, Do(a_1, s))$
3.  $Poss(If(f, a_1, a_2), s) \equiv [Holds(f, s) \wedge Poss(a_1, s)] \vee [\neg Holds(f, s) \wedge Poss(a_2, s)]$

4.  $Poss(a_1 \cdot a_2, s) \equiv Poss(a_1 + a_2, s) \equiv Poss(a_1 \oplus a_2, s) \equiv Poss(a_1, s) \wedge Poss(a_2, s)$
5.  $Poss(Loop(f, a_1), s) \equiv [Holds(f, s) \rightarrow Poss(a_1, s)] \wedge [Holds(f, Do(a_1, s)) \rightarrow Poss(a_1, Do(a_1, s))] \wedge \dots$

The foundational axioms in Definition 4.1.2 allow for calculating preconditions for composite services, given the preconditions for the participating services and the composition schema. For instance, the precondition of the *RetrieveStatus/RetrieveLocation* composition included in the process of the running example can be calculated using axiom 4 of Definition 4.1.2, as the conjunction of the preconditions of the two services.

**Definition 4.1.3** The foundational axioms for **postconditions** consist of:

1.  $State(Do(\epsilon, s)) = State(s)$
2.  $Poss(a_1; a_2, s) \rightarrow State(Do(a_1; a_2, s)) = State(Do(a_2, Do(a_1, s)))$
3.  $Poss(If(f, a_1, a_2), s) \rightarrow [Holds(f, s) \wedge State(Do(If(f, a_1, a_2), s)) = State(Do(a_1, s))] \vee \neg[Holds(f, s) \wedge State(Do(If(f, a_1, a_2), s)) = State(Do(a_2, s))]$
4.  $Poss(a_1 \cdot a_2, s) \rightarrow State(Do(a_1 \cdot a_2, s)) = State(Do(a_2, s)) + \theta_1^+ - \theta_1^- = State(s) + \theta_2^+ - \theta_2^- + \theta_1^+ - \theta_1^-$
5.  $Poss(a_1 + a_2, s) \rightarrow [State(Do(a_1 + a_2, s)) = State(s) + \theta_1^+ - \theta_1^-] \vee [State(Do(a_1 + a_2, s)) = State(s) + \theta_2^+ - \theta_2^-] \vee [State(Do(a_1 + a_2, s)) = State(s) + \theta_1^+ - \theta_1^- + \theta_2^+ - \theta_2^-]$
6.  $Poss(a_1 \oplus a_2, s) \rightarrow [State(Do(a_1 \oplus a_2, s)) = State(s) + \theta_1^+ - \theta_1^-] \oplus [State(Do(a_1 \oplus a_2, s)) = State(s) + \theta_2^+ - \theta_2^-]$
7.  $Poss(Loop(f, a_1), s) \rightarrow [\neg Holds(f, s) \rightarrow (State(Do(Loop(f, a_1))), s) = State(s)] \wedge [Holds(f, s) \wedge \neg Holds(f, Do(a_1, s)) \rightarrow (State(Do(Loop(f, a_1), s))) = State(s) + \theta_1^+ - \theta_1^-] \wedge \dots$

The axioms included in Definition 4.1.3 complement the ones in Definition 4.1.2, resulting in a whole view of a composite service execution. For instance, by combining the third axioms in Definitions 4.1.2 and 4.1.3, we can express the fact that conditional execution of *EReport* and *MReport* requires only one of the two services' preconditions to be true, depending on the truth value of the condition fluent, while a successful execution leads to a state change as a result of either *EReport* or *MReport*, again depending on the truth value of the condition.

Definitions 4.1.2 and 4.1.3 can be extended in a straightforward way for compositions of more than two services. As discussed later on in Chapter 5, the incomplete axioms that refer to loops are avoided thanks to the FLUX representation of WSSL, the recursive capabilities of Prolog, as well as imposing an upper bound on the number of iterations to guarantee termination.

Apart from defining control flow for service composition, WSSL needs to account for data flow as well. The following axiom models the simplest case of routing between outputs and inputs of services:

**Definition 4.1.4** The foundational axiom for **data flow** expresses the fact that any produced output can potentially be consumed as an input from that state onward and is written as  $Holds(HasOutput(f), z) \rightarrow Holds(HasInput(f), z)$ .

Note that composition goals can be expressed using the logical expression syntax. The WSSL extension for composition that was presented in this section is the cornerstone of the composition and verification framework that is analyzed in Chapter 5.

## 4.2 Quality of Service

By definition, any WSSL term can be associated with concepts defined in a knowledge representation model, using IRI [Duerst and Suignard 2005] sequences. These IRIs can refer to concepts of any origin, including ontology-based QoS models. In this section, we define an extension for WSSL in the direction of supporting

QoS profiles. This extension presupposes the existence of an ontology-based QoS model, defined using languages such as OWL-Q [Kritikos and Plexousakis 2009b], from which the required QoS concepts are obtained. OWL-Q is an OWL-S [Martin et al. 2004] extension that provides a semantic, rich and extensible meta-model for describing QoS aspects of service specifications, which can be used by service providers to model QoS attributes. Apart from defining the WSSL extension for QoS profiles, we also formalize correctness for such profiles as well as local and global goal matching based on them.

#### 4.2.1 WSSL QoS Profiles

In order to support QoS-aware specifications, the WSSL definition listed in Section 3.2.2 is extended to an 8-tuple by including a **quality** tuple, which represents a set of QoS profiles. Each service specification may be linked with one or more concrete service realizations. Each one of these concrete services has at least one QoS profile, and possibly more, in order to be able to model *classes of service* [Tosic et al. 2003], i.e. different QoS levels provided to consumers according to their needs and/or what they can afford. Each QoS profile is itself a set of WSSL logical expressions, expressing QoS offerings as constraints, following the form  $\langle \text{QoS term} \rangle \langle \text{comp-operator} \rangle \langle \text{value} \rangle$ , where a QoS term may be an attribute or a metric. Since they follow the WSSL logical expression syntax, constraints can be combined using any of the supported logical connectives (e.g.,  $\neg$ ,  $\wedge$  and  $\vee$ ).

To be able to effectively express constraints, the alphabet of WSSL logical expressions is extended in two ways. First, a new sort named *DECIMAL* is introduced in order to represent decimal QoS values, since they cannot be expressed by the already included sorts. Second, a set of comparison operators is included for the new sort, in addition to equality which is already included for all sorts in the initial alphabet. The comparison operators are  $\{<, \leq, \neq, \geq, >\}$ . These operators are essentially binary predicates, mapping term pairs to truth values. As far as attributes are concerned, they are represented by constants. Based on this



modeling, the global constraints of the running example would, for instance, be expressed as  $owlqmodel\#cost \leq 10 \wedge owlqmodel\#throughput \geq 50$ , where *owlqmodel* is a namespace prefix referring to a namespace that links to a QoS model where the concepts of *cost* and *throughput* are defined.

QoS attributes can be distinguished into measurable and unmeasurable ones. The former can be measured using one or more QoS metrics, i.e. concepts that analyze measurement details, while the latter cannot be measured and model static information that is qualitative in nature. Values for measurable attribute constraints are expressed using constants of the *DECIMAL* sort. Accordingly, values for unmeasurable attribute constraints are represented by IRIs. For instance, robustness/flexibility, as defined in [Ran 2003] can have the value set  $\{inflexible, flexible, very-flexible\}$ , which is modeled using three IRIs that represent the possible values. Optionally, unmeasurable attribute value sets can be mapped to decimals, provided that a suitable mapping to decimal (or integer) values exists. For instance, the aforementioned value set can be mapped to set  $\{0, 1, 2\}$ .

In the same way constraints are used to express QoS offerings in WSSL specifications, they are used to express QoS goals for WSSL-based service composition. Goals can be global, referring to the entire composite process as a whole, or local, pertaining to a specific task in the process. The overall cost threshold imposed in the running example is an example of a global QoS goal. In a WSSL goal specification, local goals are explicitly associated with the name of the task to which they refer. The XML schema for WSSL/XML that is included in Appendix B takes into account specification of local and global QoS goals.

The resulting extended version of WSSL allows the specification of both functional and non-functional aspects. Achieving such levels of completeness in specifications requires a significant modeling effort from parties involved in creating and delivering an SBA, such as service designers, developers and providers. This effort can be reduced through the use of appropriate modeling tools that assist designers in completing specifications by automatically detecting missing information (e.g., produce QoS descriptions through service monitoring or testing).

It should be stressed that specification correctness is of paramount importance: composition and verification processes rely on them, hence incorrect specifications would either lead to unsolvable problems or solving a different problem.

#### 4.2.2 Correctness of QoS Profiles

The responsibility for ensuring correctness of QoS profiles (and service specifications, in general) lies with service providers. However, since one cannot assume that such profiles will always be correct, it is of practical use to detect obvious cases where correctness is violated. Violation detection depends primarily on the operators of the constraints included in the profile, as well as the associated values. The following definition formalizes QoS profile correctness violation.

**Definition 4.2.1** Given a QoS profile  $P$  containing a set  $C$  of  $N$  constraints of the form  $\langle \text{term} \rangle \langle \text{op} \rangle \langle \text{value} \rangle$ , the profile is incorrect if for any constraint  $i, j$ , with  $0 \leq i \leq N, 0 \leq j \leq N$  it holds that  $i \neq j, \text{term}_i = \text{term}_j$  and one of the following:

1.  $op_i \equiv =, op_j \equiv =$  and  $value_i \neq value_j$
2.  $op_i \equiv =, op_j \equiv \neq$  and  $value_i = value_j$
3.  $op_i \equiv =, op_j \equiv <$  and  $value_i \geq value_j$
4.  $op_i \equiv =, op_j \equiv >$  and  $value_i \leq value_j$
5.  $op_i \equiv =, op_j \equiv \leq$  and  $value_i > value_j$
6.  $op_i \equiv =, op_j \equiv \geq$  and  $value_i < value_j$
7.  $op_i \equiv <, op_j \equiv >$  and  $value_i \leq value_j$
8.  $op_i \equiv \leq, op_j \equiv >$  and  $value_i \leq value_j$
9.  $op_i \equiv \geq, op_j \equiv <$  and  $value_i \geq value_j$
10.  $op_i \equiv \leq, op_j \equiv \geq$  and  $value_i < value_j$

Note that the complete set of cases presented in Definition 4.2.1 is relevant only for measurable attributes or unmeasurable attributes whose values constitute an ordered set. For all other types of constraints, only cases 1 and 2 make sense.

### 4.2.3 Local QoS Goal Matching

In the same manner that a QoS profile is considered incorrect, we can deduce whether a constraint contained in a QoS profile violates a local QoS goal of a composition problem. The next definition formalizes local QoS goal violation. Note that by positively monotonic we refer to metrics where a greater value represents a better QoS (such as availability), while the opposite holds for negatively monotonic metrics (such as cost).

**Definition 4.2.2** Given a QoS profile containing a constraint  $S$  of the form  $\langle term_S \rangle \langle op_S \rangle \langle value_S \rangle$ , and a local QoS goal of the form  $\langle term_T \rangle \langle op_T \rangle \langle value_T \rangle$ , the goal is **not** achieved if  $term_S = term_T$  and one of the following holds:

1.  $value_S \neq value_T$  and  $op_S \equiv op_T \equiv =$
2.  $value_S = value_T$  and
  - (a)  $op_S \equiv =$  and  $op_T \equiv \neq$
  - (b)  $op_S \equiv \neq$  and  $op_T \equiv =$
3.  $value_S < value_T$ , terms refer to positively monotonic metrics and
  - (a)  $op_S \equiv =$  and  $op_T \in \{>, \geq\}$
  - (b)  $op_S \equiv >$  and  $op_T \in \{=, \neq, >, \geq\}$
  - (c)  $op_S \equiv \geq$  and  $op_T \in \{=, >, \geq\}$
4.  $value_S > value_T$ , terms refer to negatively monotonic metrics and
  - (a)  $op_S \equiv =$  and  $op_T \in \{<, \leq\}$

- (b)  $op_S \equiv <$  and  $op_T \in \{=, \neq, <, \leq\}$
- (c)  $op_S \equiv \leq$  and  $op_T \in \{=, <, \leq\}$
- 5.  $value_S \leq value_T$  and  $op_S \equiv \geq, op_T \equiv \neq$
- 6.  $value_S \geq value_T$  and  $op_S \equiv \leq, op_T \equiv \neq$
- 7. The following combinations result in failure, regardless of values:
  - (a)  $op_S \equiv \neq$  and  $op_T \in \{<, \leq, >, \geq\}$
  - (b)  $op_S \in \{<, \leq\}$  and  $op_T \in \{>, \geq\}$
  - (c)  $op_S \in \{>, \geq\}$  and  $op_T \in \{<, \leq\}$

#### 4.2.4 QoS Metrics Analysis and Aggregation

Global QoS goal matching is a bit more complicated than the local case, since the value achieved by the composite service as a whole must be aggregated before comparing it to the targeted one. In this subsection, we present a series of aggregation functions based on a thorough analysis of QoS metrics that have been referenced in literature.

The aggregation process depends on two dimensions: the composition pattern linking the services whose QoS values are aggregated, and the nature of the QoS attribute. Concerning composition patterns, we focus again on the set of patterns that was analyzed in Section 4.1: sequence, (deterministic) loop, AND-Split/AND-Join, AND-Split/DISC-Join, OR-Split/OR-Join, OR-Split/DISC-Join and XOR-Split/XOR-Join. Note that aggregation functions cannot be defined for any loop; aggregating a value depends on the knowledge of an upper bound on the number of iterations, the so-called *loop variant*. Also, we assume that no knowledge about the probability of execution paths is available; each possible choice in a composite process is considered equally probable. The aggregation functions that result from the analysis are essentially a generalization of the approach in [Rosenberg et al. 2009] in order to refer to attribute categories instead of specific attributes.

Although there is a great multitude of attributes, they can be classified in a way that drives the aggregation process. We considered QoS attributes that are included in the detailed analysis of domain-independent attributes presented in [Kritikos and Plexousakis 2009b]. We expect that any additional attributes, including domain-dependent ones, will also fall into one of the groups we define. We examine measurable and unmeasurable QoS attributes separately.

Unmeasurable attributes represent quality information that is static and can be grouped into the following three categories:

**Boolean** All attributes in this category represent properties that are either supported or not by a service. Examples of attributes presented in [Kritikos and Plexousakis 2009b] that fall in this category are: integrity (if defined as support, or lack thereof, of the ACID properties), exception handling, two-phase commit, supported standards, guaranteed messaging requirements, authentication, authorization, confidentiality, accountability, traceability and auditability, non-repudiation and safety.

**Ordered Set** This category includes attributes whose possible values form an ordered set, such as robustness/flexibility, as defined in [Ran 2003]

**Unordered Set** The third category is formed from attributes that take values from an unordered set, such as failure masking, operation semantics, server failure, data encryption, security level and service level.

For all of these groups, aggregation is pattern-independent, since it involves finding whether all services share certain characteristics (or finding the lowest-ordered one, in the case of ordered sets). Note that for unordered sets, the more features a service offers, the more likely it is to find common ones across services participating in a composite process. In cases where intersection leads to an empty set, a sub-process for which the attribute in question is more relevant could be taken into account, e.g., determining whether data encryption is supported only in the precise part of a process that manages sensitive data.

Measurable attributes can be measured using one or more QoS metrics and are grouped into the following five categories:

**Reputation-like** Aggregation for attributes in this category involves calculating the mean value, to provide an average value of what is offered. Note that it is also possible, albeit less likely, that a min-max range for reputation is demanded. Apart from reputation, which lends its name to the category, the attribute of bandwidth shares that same nature.

**Throughput-like** This category contains attributes that are similar to throughput, in the sense that aggregation corresponds to finding the process bottleneck. Apart from throughput, mean time between failure is also in this category.

**Cost** This is a single-attribute category, since cost is the only one that is additive in both sequential and AND-\* patterns, due to the fact that aggregation must consider the cost of all services that may begin execution. It should be mentioned that cost aggregation is directly dependent to the cost model of each service provider and may deviate from the general aggregation rules that we define here, because of specific aspects of the associated cost model. For instance, there may be a monthly fee for using a service, regardless of the number of executions.

**Availability-like** The fourth category contains probabilistic-valued attributes, such as availability, the aggregation of which involves taking the product of all values for sequences. Continuous availability and completeness also fall into this category.

**Time** The fifth and final category groups attributes of temporal nature; aggregation involves taking the sum for sequential and the maximum for parallel execution, since these represent the worst case scenario. This category contains the following attributes: response time, latency, execution time,

transaction time, network delay, delay variation, queue delay time and accessibility.

Out of these five categories, the first two are also pattern-independent: whether services are executed in sequence or in parallel does not affect calculating averages or minimums. Aggregation functions are summarized in Table 4.1, with  $x_a$  denoting the aggregated value and  $x_i$  denoting the advertized values for services involved in a composition following a particular pattern.

Deterministic loops are essentially a sequence of length equal to the maximum number of iterations, defined by the loop variant. The type of join following an AND-Split is also irrelevant to aggregation; considering only  $m$  out of  $n$  paths at the end does not change the fact that the worst case scenario must be considered for aggregation. Finally, OR and XOR patterns generally yield a min-max pair of values, since we cannot know, at design time, which of the branches are going to be selected. Note that some cases, such as maximum availability or minimum execution time, may not always be interesting but they can be useful when dependencies exist between attributes (e.g., cost values that are dependent on the range of execution times offered).

### 4.3 Partial Observability

The final WSSL extension that we propose involves the aspect of partial observability with regard to WSSL states. Partial observability arises due to executing actions that do not have a deterministic outcome. Such actions may be ones that have a state update axiom with more than one alternatives, using the generalized form of Eq. 3.1. Partial observability is also closely related to service composition, when composite services employ a conditional control construct or a non-deterministic loop.

To deal with partial observability, we once again turn to the fluent calculus foundations of WSSL. There are two levels of handling this issue. The lower level is based on the generalized notion of an incomplete fluent calculus state, as defined

Table 4.1: Categorization and Aggregation of QoS attributes

QoS Attribute Categories	Composition Patterns		
	Sequence Det. Loop	AND-AND AND-Disc	OR-OR OR-Disc XOR-XOR
Measurable	Temporal	$x_a = \sum_{i=1}^n x_i$	$x_a = \max\{x_1, \dots, x_n\}$
	Probabilistic	$x_a = \prod_{i=1}^n x_i$	$x_a = \{ \min\{x_1, \dots, x_n\}, \max\{x_1, \dots, x_n\} \}$
	Cost	$x_a = \sum_{i=1}^n x_i$	$\max\{x_1, \dots, x_n\}$
	Reputation	$x_a = \text{avg}\{x_1, \dots, x_n\}$	
Unmeasurable	Throughput	$x_a = \min\{x_1, \dots, x_n\}$	
	Boolean	$x_a = \begin{cases} false, & \exists i \cdot (x_i = false) \\ true, & otherwise \end{cases}$	
	Ordered Set	$x_a = \min\{x_1, \dots, x_n\}$	
	Unordered Set	$x_a = \prod_{i=1}^n x_i$	



in Chapter 3 of [Thielscher 2005b] and requires a minimal set of modifications to the language. The higher level of handling partial observability involves modeling what it means to *know* that a fluent holds, based on a simplified version of knowledge modeling, as defined in Chapter 5 of [Thielscher 2005b].

### 4.3.1 Incomplete States

So far, states are specified as a composition of a finite number of fluents:  $z = f_1 \circ \dots \circ f_n$ . However, we may not have complete knowledge of a state, especially in the case of initial states with regard to service composition. In this case, the only way to specify a state is through constraints, resulting in *incomplete states*. While complete states are finite, incomplete states may correspond to an infinite set of possible states.

Constraints that take part in an incomplete state definition can be of various types. [Thielscher 2005b] defines the following four:

- Negation constraints of the form  $\neg Holds(f, z)$ , expressing that a fluent does not hold in a state.
- Universal quantification constraints of the form  $(\forall y)\neg Holds(f, z)$ , denoting that a fluent does not hold in a state for all values of a variable  $y$ , referring to one of the fluent's arguments.
- Disjunction constraints of the form  $Holds(f_1, z) \vee \dots \vee Holds(f_n, z)$  with  $n \geq 1$ , expressing the knowledge that one or more of a set of fluents hold.
- Arithmetic constraints: in case any arguments of fluents are encoded using numbers, then arithmetic constraints on the values of these arguments can be expressed.

To take into account incomplete states, state update axioms are slightly modified, leading to the following form:

$$Poss(A(x), s) \rightarrow (\exists y)(\Delta(s) \wedge (\exists z)(State(Do(A(x), s)) = \tau \circ z + \theta^+ - \theta^- \wedge \Phi))$$

where  $\tau \circ z \wedge \Phi$  is the general form of an incomplete state, with  $\tau$  a set of fluents and  $\Phi$  a set of constraints. Essentially, we are certain that the fluents in  $\tau$  hold in state  $z$ ; for any other fluent to hold in  $z$ , it must satisfy the set of constraints  $\Phi$ . For instance, in the running example, we would like to express the fact that a request for assistance can be received from the user either by sending an SMS or by calling, or both. This can be accomplished by including the constraint  $Holds(HasInput(sms), ?z\_in) \vee Holds(HasInput(call), ?z\_in)$  in the definition of the initial state.

### 4.3.2 Knowledge States

Modeling of incomplete states allows for a way to handle partial observability without any major adjustments to the existing language foundations. Essentially, handling partial observability is delegated to handling constraints, which is a matter of the way a fluent calculus language is implemented. A more formal way of dealing with this issue is through the modeling of *knowledge states*, acting as a generalization of incomplete state modeling, without contradicting it.

A knowledge state (or *possible state*) is any state of the world that may be true according to our state knowledge. There is an inverse relationship between the number of possible states and the amount of state knowledge. If we have complete state knowledge, then there is only one knowledge state: the actual one. On the other end of the spectrum, if we have no knowledge about the current state, then all conceivable states are possible. An incomplete state knowledge yields several possible states, based on a set of constraints.

In order to represent knowledge states, a new predefined predicate is introduced,  $KState(s, z)$ . Then, a knowledge state is defined as follows:

**Definition 4.3.1** A *knowledge state* is a formula  $KState(s, z) \equiv \Phi(z)$ , defining a possible state  $z$  in a situation  $s$ , under a set of constraints expressed in state formula  $\Phi$ .

Based on Definition 4.3.1, [Thielscher 2005b] defines a helpful macro for what it

means to know a state property: all possible states entail it.

**Definition 4.3.2** A knowledge expression  $\phi$  is known in a situation  $s$  based on the following:  $Knows(\phi, s) \stackrel{def}{=} (\forall z)(KState(s, z) \rightarrow HOLDS(\phi, z))$  where  $\phi$  may consist of fluents, stateless Poss predicates of the form  $Poss(a)$  and atoms without state or situation terms, while  $HOLDS(\phi, z)$  is obtained by replacing in  $\phi$  all fluents with a Holds expression of the form  $Holds(f, z)$  and adding state  $z$  to all Poss predicates.

The simplest knowledge expression consists of a single fluent  $f$ . Based on Definition 4.3.2, knowing  $f$  in situation  $s$  means that the following is true:

$$Knows(f, s) \stackrel{def}{=} (\forall z)(KState(s, z) \rightarrow Holds(f, z))$$

In other words,  $f$  holds in all possible states at situation  $s$ .

Knowledge states are then used to redefine state update axioms to take possible states into account. At this point, we diverge from the original fluent calculus knowledge modeling proposed in [Thielscher 2005b]. This is due to the fact that this modeling is based on the differentiation between physical and cognitive effects, which is relevant to autonomous robotic agents: they can act upon the environment with physical actions or gain knowledge about it through sensors. However, the service case is much simpler: knowledge acquisition is neither a physical nor a cognitive effect, at least in the way defined for agents. Hence, we choose to simplify state update with knowledge, by keeping the initial way of modeling effects: an addition and/or subtraction of fluents.

**Definition 4.3.3** A knowledge update axiom for an action  $A(x)$  is a formula

$$Poss(A(x), s) \rightarrow (\exists y)(\Delta(s) \wedge KState(Do(A(x), s), z') \equiv (\exists z)(KState(s, z) \wedge z' = z + \theta^+ - \theta^-))$$

A knowledge update axiom with ramifications for the same action is a formula

$$Poss(A(x), s) \rightarrow (\exists y)(\Delta(z) \wedge KState(Do(A(x), s), z') \equiv (\exists z)(KState(s, z) \wedge Ramify(z, \theta^+, \theta^-, z', Do(A(x), s)))$$

Essentially, state update is applied as before but on knowledge states, expressed by *KState* instead of complete states. The states which are possible after a service execution are directly related to the possible states prior to execution, with the addition or subtraction of fluents, including the application of any causal relationship that may be relevant. Thus, handling partial observability amounts to replacing state update axioms with their knowledge update counterparts, expressing initial states as knowledge states instead of complete, ground states and including any other domain knowledge as restrictions on knowledge states instead of ground states.

# Chapter 5

## Implementation

### Contents

---

<b>5.1</b>	<b>Implementing WSSL in FLUX</b>	<b>120</b>
5.1.1	FLUX kernels	120
5.1.2	Domain Encoding	122
5.1.3	WSSL Verification Tool	124
5.1.4	Planning	125
<b>5.2</b>	<b>WSSL/CVF: Composition and Verification Framework</b>	<b>131</b>
5.2.1	Functional Composition and Verification	134
5.2.2	Specification-based Functional Discovery	136
5.2.3	Extended Plan Pruning and Ranking	138
5.2.4	QoS-based Selection	140
5.2.5	Framework Use	143
5.2.6	Limitations	144

---

In this chapter, we demonstrate how WSSL and its extensions can be employed in order to realize service-related activities in addition to service specification, namely automated service composition and verification. To that end, an implementation of WSSL is proposed, using the logic programming language FLUX and based on fluent calculus foundations. Then, WSSL/CVF, a complete QoS-aware service composition and verification framework is presented and analyzed in depth.

The framework utilizes WSSL's implementation in FLUX and exploits the unique features of the language that allow for rich service specifications and support for solutions to the frame, ramification and qualification problems.

## 5.1 Implementing WSSL in FLUX

FLUX (FLUent eXecutor), defined and analyzed in [Thielscher 2005a], is a programming language and system that offers an implementation of the fluent calculus in logic programming with constraint handling. FLUX has a restricted expressiveness, employing Prolog with Constraint Handling Rules (CHR), that enables an excellent computational behavior, offering reasoning in linear time with regard to the size of state representation. It is expressive enough, however, to be able to support any WSSL specification. As such, it is an excellent choice for implementing WSSL as a logic programming language in order to use it as a basis for service validation, verification and composition.

For our purposes, FLUX programs consist of two layers: a kernel program that includes clauses that realize reasoning based on the fluent calculus semantics and, on top of it, the encoding of the service domain, derived from the WSSL specifications for each service. We examine these two layers separately.

### 5.1.1 FLUX kernels

We define two different kernels which are customized versions of the complete FLUX kernel<sup>1</sup> that is defined and analyzed in [Thielscher 2005b]. The first kernel is a basic version that supports state specification and update, as well as the ramification and qualification problems; the second kernel is a modification of the first in order to support incomplete and knowledge states, as defined in Section 4.3. The complete Prolog code for both kernels, including all modifications analyzed in this section, as well as the FLUX encoding for the running example,

---

<sup>1</sup>The Prolog code for this kernel is available online at <http://www.fluxagent.org>. Copyright (c) 2000 by The Dresden University of Technology, Dresden, Saxony, Germany. All Rights Reserved.

can be found in Appendix C.

The *basic kernel* contains the following definitions without any modification from the original FLUX kernel:

**holds(F,Z)** Succeeds when fluent F is part of state Z. An auxiliary ternary variant is also included, with the third argument denoting the state minus the fluent.

**minus\_(Z1, ThetaN, Z2)** Defined using the ternary Holds; succeeds if Z2 is the state that is derived if the list of fluents ThetaN is removed from Z1.

**plus\_(Z1, ThetaP, Z2)** Succeeds if Z2 is the state that is derived if the list of fluents ThetaP is added to Z1.

**update(Z1, ThetaP, ThetaN, Z2)** Succeeds if state Z2 equals to state Z1, after removing fluents in list ThetaN (using *minus\_/3*) and adding fluents in list ThetaP (using *plus\_/3*). This clause provides the solution to the inferential aspect of the frame problem, as detailed in [Thielscher 2000].

Additionally, the following simplified clause for the *ramify* macro is included:

```

1 ramify(Z1,ThetaP,ThetaN,Z2) :-
2   update(Z1,ThetaP,ThetaN,Z),
3   (causes(Z,ThetaP,ThetaN,Z2,_,_) ; Z2=Z).
```

Listing 5.1: Simplified version of *ramify/4*

Essentially, following the application of state update using the *update* clause, either a causal relationship is found and applied or no other modifications are made. Causal relationships are expected to be expressed using a *causes* clause with the above format, as defined later on. In the unlikely case that ramification chains of length more than 1 are required to be modeled, then either a new version of *ramify* must be defined, customized to the desired length, or the original version can be reused, since it is defined assuming ramification chains of arbitrary length.

The *full kernel* handles incomplete states in two complementary ways, following the original FLUX kernel. First, incomplete FLUX states are modeled as lists

with a variable tail:  $[f_1, \dots, f_n \mid z]$ . Then, constraint logic programming is used in order to handle constraints. A set of rules is defined for each type of constraints (negation, disjunction and universal quantification constraints), while arithmetic constraints are handled by the underlying system libraries (more on the underlying system in Section 5.2).

The following definitions of the original kernel are also included:

**cancel(F, Z1, Z2)** Succeeds when state Z2 is state Z1 with all knowledge of fluent F canceled

**knows(F, Z)** fluent F is known to hold in state Z

**knows\_not(F, Z)** fluent F is known not to hold in state Z

Definitions for knowledge (or lack thereof) are realized in an elegant way using negation as failure. This allows them to have the same computation cost as checking whether a fluent holds or not in a state.

### 5.1.2 Domain Encoding

While the aforementioned FLUX kernels are domain-independent, modeling a specific set of services requires a domain-specific encoding based on WSSL specifications. Table 5.1 offers a mapping from WSSL elements to FLUX clauses (written in Prolog). Due to the declarative nature of FLUX, fluent calculus axioms become program statements. Note that inputs and preconditions are combined to a single `poss` clause and outputs and postconditions to a single `state_update` one to preserve correct service execution semantics.

The  $\hat{\phantom{x}}$  denotes that the particular clause has been suitably translated in FLUX. Note that for efficiency reasons, translation of postconditions omits the verification of the precondition, assuming that services are executed only in states where such an execution is possible. If the user wants to verify them, then they should be included in the query expression. Also, the situation argument is suppressed



Table 5.1: Mapping from WSSL to FLUX

WSSL	FLUX
$ Holds(HasInput(x), z) $ $ Poss(A(x), z) \equiv \Pi(z) $	$ poss(a(x), Z) :- $ $ holds(hasinput(x), Z), \hat{\Pi}(z). $
$ Holds(HasOutput(x), z) $ $ Poss(A(x), z_1) \rightarrow \Delta(z_1) \wedge $ $ State(Do(A(x), z_1)) = $ $ State(z_1) - \theta^- + \theta^+ $	$ state\_update(Z\_1, a(x), Z\_2) :- $ $ \hat{\Delta}(Z\_1), $ $ update(Z\_1, [\theta^+, hasoutput(x)], $ $ \theta^-, Z\_2). $
$ Ramify(z_1, \theta^+, \theta^-, z_2, Do(A(x), s)) $	$ ramify(Z\_1, \theta^+, \theta^-, Z\_2) $
$ \Gamma \rightarrow Causes(z_1, p_1, n_1, z_2, p_2, n_2, s) $	$ causes(Z\_1, P\_1, N\_1, Z\_2, P\_2, N\_2) :- \hat{\Gamma}. $
$ z_0 = f_1 \circ \dots \circ f_n $	$ init(Z\_0) - Z\_0 = [f_1, \dots, f_n]: $

in the translation of *Ramify* and *Causes*. As far as accidents are concerned, we deviate from the original FLUX kernel and simplify the representation of accidental updates, modeling them as disjuncts of the `state_update` clause of the associated service. Hence, the `state_update` clause contains not only the normal case, but also all accident cases, in the order they are expressed in the initial representation.

The `init` predicate allows for the specification of initial states. Note that states are modeled as Prolog lists. This leads to a limitation of FLUX in relation to the fluent calculus, since such lists are ordered, while in the fluent calculus states are unordered sets of fluents. This means that a single fluent calculus state corresponds to multiple FLUX states (one for each possible ordering of the fluents contained).

It should be noted that when translating a WSSL specification to a FLUX program, fluent arguments are almost always translated to Prolog constants. For instance,  $HasInput(request)$  of *RetrieveLocation* translates to `hasinput(request)`. If it was translated to `hasinput(REQUEST)` (all capitals denote a Prolog variable), then answering any query, for example `poss(RetrieveLocation, Z)` would also involve attempting to bind `REQUEST`, which is not our intention: `request` represents an input message that needs to be available to a service before exe-

cution, hence the poss query should just make sure that `hasinput(request)` is part of the binding of variable `Z`.

FLUX also offers the ability to express domain constraints that determine whether a state is consistent or not. This is achieved by adding a clause with the predicate `consistent(Z)` as its head and the FLUX encoding of constraints as its body, with constraints referring to state variable `Z`. Whenever a state needs to be checked for its consistency, the `consistent` predicate is called, with a variable referring to the state in question as its argument.

The translation process from WSSL to FLUX can be trivially proven to be sound since it follows the soundness of the encoding of the fluent calculus in FLUX. For instance, the FLUX clause `B: holds(f, Z)` is the translation of a WSSL expression `A: Holds(f, z)` for a fluent `f` and a state variable `z`. `B` is indeed the FLUX encoding of the fluent calculus macro `Holds`, hence the translation is sound and the same applies to all WSSL element translations in Table 5.1. A detailed correctness analysis for the FLUX encoding of the fluent calculus is carried out in Section 6.1.

### 5.1.3 WSSL Verification Tool

To exploit the expressive capabilities of WSSL, we implemented a verification tool that takes a WSSL specification as input and allows the user to express goals that are verified against the specification. The verification process involves 3 basic steps. First, a WSSL document (written in the WSSL/XML syntax) is translated into a FLUX program. Afterwards, the tool expects a WSSL goal to be expressed by the user, which is also translated in FLUX (the user can also express FLUX queries directly). The final step involves automatically loading the program in the *ECL<sup>i</sup>PS<sup>e</sup>* Constraint Programming System [Apt and Wallace 2007], verifying the goal and returning the results to the user. *ECL<sup>i</sup>PS<sup>e</sup>* is an open-source software system for developing and deploying constraint programming applications and includes support for CHR as well as several constraint solving libraries and is also backwards-compatible with Prolog.

Translating a WSSL/XML document to a FLUX program is a fairly straightforward process and is based on the mapping defined in the previous section. The main issue to be tackled is the fact that information is contained in a slightly different order in the XML document than it is expected in a FLUX program. Thus, information must be gathered from the XML document and rearranged so that the produced FLUX program is compilable using a Prolog interpreter. Algorithm 1 creates a parser that takes a WSSL/XML document as input and produces the equivalent FLUX program.

The verification tool, including the parser detailed in Algorithm 1, was implemented as a Java program with a Swing interface, employing the Java API for XML Processing (JAXP), as well as the Java-*ECL<sup>i</sup>PS<sup>e</sup>* interface library. Evaluation results for the translation process can be found in Section 7.1.1. An example FLUX query fed to the verification tool, given the service specification of *ReceivePay* in Tables 3.1 and 3.2 is:

```
poss(receivepay, Z), state_update(Z, receivepay, Z1).
```

which would yield the result

```
Z=[hasinput(creditcard), solved(status, location)] and
Z1=[Z, hasoutput(invoice), paycompleted(payform)]
```

This query essentially requests the valid states before and after a successful execution of *ReceivePay*. If the query was posed in the form

```
poss(receivepay, Z), Z=[solved(status, location)]
```

then the answer would be no, since the required input for *ReceivePay* is not available at state Z.

#### 5.1.4 Planning

Apart from service verification, the implementation of WSSL using FLUX can also be used in service composition. In the series of definitions that follows, we

**Algorithm 1:** Translating a WSSL/XML document to a FLUX program

```

input : A WSSL specification using the WSSL/XML syntax
output: The equivalent FLUX program

inpList  $\leftarrow$  GetInpElems(), preList  $\leftarrow$  GetPreElems();
foreach element i of inpList do get name of input i;
foreach element pr of preList do
  if pr == 0 then
    | find input state variable name;
    | write action precondition head and the HasInput part of the body;
  end
  else
    | translate logical expressions and write rest of body;
  end
end

outpList  $\leftarrow$  GetOutpElems(), postList  $\leftarrow$  GetPostElems();
foreach element o of outpList do get name of input o;
foreach element po of postList do
  foreach disjunct d in po do
    | if d is an accident case then
    | | translate logical expression and write accident disjuncts;
    | end
    | else
    | | translate logical expression and write normal disjuncts;
    | end
  end
end

end

write state_update axiom;
foreach element cr of crList do translate causal relationship;

```

formalize WSSL service composition as planning, based on planning in the fluent calculus with FLUX (see Chap. 6 in [Thielscher 2005b]).

**Definition 5.1.1** A WSSL planning problem is defined as the problem of reaching a goal state defined by a state formula  $\Gamma(z)$ , starting from an initial state defined by a state formula  $I(z)$ . A WSSL plan is a sequence  $\alpha_1, \dots, \alpha_n$  of service executions, with  $n \geq 0$ . The plan is a solution to the problem iff the following holds:  $Poss([\alpha_1, \dots, \alpha_n], \Phi(z)) \wedge \Gamma\{z/State(Do([\alpha_1, \dots, \alpha_n], I(z)))\}$

A planning problem is encoded in FLUX in a sound and complete way using the following two clauses:

$$P(z, p, z) \Leftarrow Goal(z), p = [] \text{ and} \\ P(z, [a|p], z_n) \Leftarrow Poss(a, z), StateUpdate(z, a, z_1, []), P(z_1, p, z_n)$$

These clauses state that if we are at the goal state, then the solution is the empty plan, otherwise the plan is constructed recursively with a sequence of actions, until the goal state is reached. For instance, a planning problem encoding for the scenario of the running example is:

$$AssistPlan(z, p, z) \Leftarrow Holds(Solved(status, location), z), \\ Holds(PayCompleted(payform), z), Holds(HasOutput(report), z), p = [] \\ AssistPlan(z, [a|p], z_n) \Leftarrow \\ Poss(a, z), StateUpdate(z, a, z_1, []), AssistPlan(z_1, p, z_n).$$

Definition 5.1.1 and the accompanying planning problem encoding have two major drawbacks. First, they do not take into account the issues of termination and computational complexity, since they define an exhaustive systematic search through all executable sequences of actions. The more actions are available at a given state, the less computationally feasible it is to find a solution to the planning problem. The second drawback is that they can produce only sequential compositions, disregarding any other control construct, such as conditional or parallel composition. The first step towards handling both issues is introducing heuristics, only considering plans that follow a more specific encoding.

**Definition 5.1.2** A heuristic encoding of a WSSL planning problem is defined as a FLUX program  $P_{plan}$  defining a predicate  $P(z, p, z_n)$  that describes the problem of reaching a goal state  $\Gamma(z)$ , starting from an initial state  $\Phi(z)$ . The encoding is sound iff the following holds: for every computed answer  $\theta$  to the FLUX query  $\Leftarrow \Phi(z) \wedge P(z, p, z_n), p\theta$  is a solution to the planning problem and  $Poss(p\theta, \Phi(z)) \wedge \Gamma\{z/State(Do(p\theta, \Phi(z)))\}$ .

In order to consider plans more complex than sequences of services, heuristic encodings need to include control construct definitions. Based on Definitions 4.1.2 and 4.1.3, we extend the FLUX kernels that we defined with clauses that support specification of fundamental control constructs:

```

1  poss_if(F, A, B, Z) :- holds(F,Z), poss(A,Z) ; not_holds(F,Z), poss(B,Z), A\==B.
2  poss_and(A, B, Z) :- poss(A, Z), poss(B, Z), A \== B, A @< B.
3  poss_or(A, B, Z) :- poss(A, Z), poss(B, Z), A \== B, A @< B.
4  poss_xor(A, B, Z) :- poss(A, Z), poss(B, Z), A \== B, A @< B.
5  poss_loop(F,K,A,Z) :- K\==0, (holds(F, Z) -> poss(A, Z)),
6                          update(Z, A, Z_PR), poss_loop(F, K-1, A, Z_PR).
7
8  state_update_if(Z, F, A, B, Z_PR) :- holds(F, Z), state_update(Z, A, Z_PR) ;
9                          not_holds(F, Z), state_update(Z, B, Z_PR).
10 state_update_and(Z,A,B,Z_PR) :- state_update(Z,A,Z_1), state_update(Z_1,B,Z_PR).
11 state_update_or(Z,A,B,Z_PR) :- state_update(Z,A,Z_1), state_update(Z_1,B,Z_PR) ;
12                          state_update(Z,A,Z_PR) ; state_update(Z,B,Z_PR).
13 state_update_xor(Z,A,B,Z_PR) :- state_update(Z,A,Z_PR),\+ state_update(Z,B,Z_PR);
14                          state_update(Z,B,Z_PR), \+ state_update(Z,A,Z_PR).
15 state_update_loop(Z,F,K,A,Z_PR) :- not_holds(F,Z) -> Z_PR=Z ; K\==0,
16                          (holds(F,Z), update(Z,A,Z_1), not_holds(F,Z_1)) ->
17                          state_update(Z,A,Z_1), state_update_loop(Z_1,F,K-1,A,Z_PR).

```

Listing 5.2: Control flow clauses

Note that sequence needs no special foundational axioms, as it is simply modeled using Prolog comma sequences. There are a number of notable differences

between Definitions 4.1.2 and 4.1.3 and their implementation in FLUX/Prolog in Listing 5.2:

- Lines 1-4: The `poss` clauses for conditional and parallel composition make sure that services `A` and `B` are different ( $A \neq B$ ). This avoids meaningless compositions of the same service, which are deemed valid in Prolog, provided their preconditions hold at `Z`.
- Lines 2-4: The `poss` clauses for parallel composition also include the clause `A@<B` which demands that the two services in parallel are encountered in an alphabetical order. This is included in order to make sure that only one of `A <op> B`, `B <op> A` succeeds, otherwise two identical composition schemas are regarded as different by Prolog interpreters.
- Lines 5-6 and 15-17: In the case of loops, it is necessary to impose an upper bound  $K$  on the number of iterations, to avoid non-terminating executions. Both of these clauses employ recursion to implement loop semantics.
- Line 10: State update in AND-Split/Join is expressed as in the case of sequence, since the resulting state update is the same regardless of whether the services were executed in sequence or in parallel. In accordance with the discussion in Section 4.1.1, we assume that the two services are independent, i.e. one does not negate an effect of the other. In case where the two services are not independent, one needs to model both alternatives by adding the disjunct `state_update(Z,B,Z_1), state_update(Z_1,A,Z_PR)`.
- Lines 11-12: OR semantics are approximated, in the sense that we attempt first to prove that both services have executed successfully and only if this fails, do we consider the alternatives of only `A` succeeding and only `B` succeeding, in that order.
- Lines 13-14: XOR semantics are expressed by requiring that only one of the two service state updates is provable. This means that racing behavior is not modeled; only pure XOR semantics are.

It should also be noted that the `not_holds/2` predicate, used in the definition of conditionals and loops, is implemented as a negative constraint. Listing 5.3 shows one possible heuristic encoding for the composition problem of the running example:

```

1 assist_plan (Z,[A|P],Z_PR) :- A1=receivesms, A2= receivecall ,
2   A=xor(A1,A2), poss_xor(A1,A2,Z), state_update_xor(Z,A1,A2, Z_1),
3   assist_plan1 (Z_1,P,Z_PR).
4 assist_plan1 (Z,[A|P],Z_PR) :- A1= retrievelocation , A2= retrievediagnostics ,
5   A=and(A1, A2), poss_and(A1,A2,Z), state_update_and(Z,A1,A2,Z_1),
6   assist_plan2 (Z_1,P,Z_PR).
7 assist_plan2 (Z,[A|P],Z_PR) :- A=findmech, poss(A,Z),
8   state_update (Z,A,Z_1), assist_plan3 (Z_1,P,Z_PR).
9 assist_plan3 (Z,[A|P],Z_PR) :- A=receivepay, poss(A,Z),
10  state_update (Z,A,Z_1), assist_plan4 (Z_1,P,Z_PR).
11 assist_plan4 (Z,A,Z_PR) :- F=req_deliv (report), A1=ereport, A2=mreport,
12  A=if(F,A1,A2), poss_if (F,A1,A2,Z), state_update_if (Z,F,A1,A2,Z_PR).

```

Listing 5.3: Heuristic encoding for the running example

Executing the FLUX query `assist_plan([callcenterup, gpsactive(user1), systemactive(vehicle1), req_deliv(report1)], P, Z_PR)`. will yield a plan following the composite process of Fig. 2.1. Additionally, if we employ the full FLUX kernel, defined earlier, queries with incomplete initial states can also be handled. For instance, we may exclude `req_deliv(report1)` (which corresponds to the user wanting the report delivered by traditional mail) from the definition of the initial state and the planner will produce plans that assume either of the two cases, with or without that fluent.

### Data Flow

The implementation in FLUX of the data flow axiom defined in Section 4.1.4 requires a new version of the `plus_/3` clause in both FLUX kernels. The new defi-



nitions are shown in Listing 5.4. In both cases, if the fluent list that is given as the second argument of `plus_contains` a *HasOutput* fluent, then the equivalent *HasInput* is also added to the final state.

```

1 plus_(Z, [], Z).
2 plus_(Z, [F|Fs ], Zp) :-
3   (\+ holds(F, Z) -> Z1=[F|Z ],(F=hasoutput(N) -> Z2=[hasinput(N)|Z1] ; Z2=Z1);
4   holds(F, Z), Z2=Z), plus_(Z2, Fs, Zp).
5
6 plus_(Z, [], Z).
7 plus_(Z, [F|Fs ], Zp) :-
8   (\+ holds(F, Z) -> Z1=[F|Z ],(F=hasoutput(N) -> Z2=[hasinput(N)|Z1] ; Z2=Z1);
9   \+ not_holds(F, Z) -> Z2=Z
10      ; cancel(F, Z, Z3), not_holds(F, Z3), Z2=[F|Z3 ], plus_(Z2, Fs, Zp).

```

Listing 5.4: Modified versions (basic and full) of `plus` to support data flow

Apart from this default, domain-independent data flow rule, one may also want to express domain-dependent links between input and output variables. This can be easily expressed using ramifications: the fact that a service produces a certain output triggers a ramification of having a certain, differently-labeled input available. Such customized rules need to be part of service specifications in order to be usable in composition scenarios. An evaluation of the WSSL planner using *ECL<sup>i</sup>PS<sup>e</sup>* is presented in Section 7.1.2.

## 5.2 WSSL/CVF: Composition and Verification Framework

WSSL planning, as was defined in the previous section, can be used as a basis for an integrated composition and verification framework that satisfies the complete set of requirements for automated service composition that was defined in Section 2.4.1, as well as exploiting the benefits of relying on WSSL service specifications. More specifically:

**Representation completeness** Employing WSSL specifications for all services

considered by the framework allows for descriptions and composition goals that do not suffer from the frame problem, take ramifications into account, while also being able to verify the resulting compositions and provide explanations for unexpected observed behavior. Also, WSSL specifications allow for behavioral state-based matchmaking between service candidates and functional goals. All these aspects are realized via the customized FLUX kernels we defined to implement WSSL and its extensions.

**Automation** This requirement is realized via WSSL-based planning and discovery; both processes are automated by applying logic programming techniques on FLUX programs.

**Dynamicity** WSSL specifications may be linked to specific service endpoints (by means of the grounding attribute in the *service* part), but this is not mandatory. Thus, a WSSL-based composition framework yields abstract plans which can then be concretized through suitable service discovery mechanisms. Since the plans are essentially compositions of service specifications, concretization can be altered in a dynamic way by discovering alternative services that still conform to the specifications.

**Semantic capabilities** WSSL specifications are semantically rich, containing inputs, outputs, preconditions and postconditions, while all elements are associated with an IRI [Duerst and Suignard 2005], which can link them to any available ontology. It should be noted that we use ontologies as shared vocabularies and not to perform reasoning-based composition and discovery; these tasks are realized using logic programming in FLUX.

**QoS-Awareness** The extension of WSSL to support QoS profiles, defined in Section 4.2, assists in realizing QoS-awareness in service composition, in association with suitable QoS-aware service discovery algorithms such as the ones defined in [Kritikos and Plexousakis 2009a;b, Mello Ferreira et al. 2009].

**Non-determinism** The extension of WSSL to support composition, defined in

Section 4.1, is implemented in FLUX and includes non-deterministic control constructs such as conditional execution and loops.

**Partial Observability** This requirement is addressed by the third and final WSSL extension, defined in Section 4.3 and also implemented in FLUX, allowing for incomplete state specification, update and reasoning, as well as the definition of knowledge (or lack thereof) with regard to a fluent.

**Domain independence** WSSL specifications are not dependent on any single domain; they can be used to specify any service.

**Correctness** The strong logic foundations of WSSL in the fluent calculus allow for the development of effective verification techniques, which are realized by posing FLUX Prolog queries in relation to the property we are attempting to prove.

**Scalability** The efficiency results of running FLUX queries in [Thielscher 2005a] indicate a good starting point for scalability aspects. Scalability evaluation is analyzed in Chapter 7.

The main functionality of WSSL/CVF is illustrated in Fig. 5.1 and analyzed in the rest of this chapter. Functional composition is realized through WSSL planning and produces a set of abstract plans that achieve functional composition goals. Verification queries can also be posed with regard to the resulting abstract plans. Each task in the abstract plans produced by the planner is associated with a set of services through functional discovery, yielding so-called *extended* plans. Then, pruning and ranking is performed for these plans by attempting to satisfy local QoS goals. Finally, QoS-based selection based on global QoS goals is executed for the top-ranked plan; if no solution is found, selection is performed for the second-ranked one and so on.

We adopt the two-phase approach for QoS-awareness that is more common in the associated literature, performing functional and non-functional composition separately. The main reason behind this decision is to split the composi-

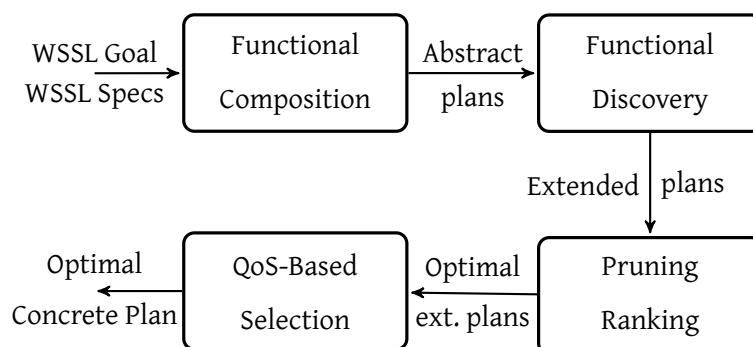


Figure 5.1: Overview of WSSL/CVF

tion problem, allowing for a more focused approach on the functional and non-functional cases; this also avoids the increase in complexity when all aspects are considered simultaneously, since that would lead in more severe cases of backtracking in planning. Separating functional and non-functional composition also allows the framework to produce abstract processes, where each task is only associated with a specification and not a concrete service, by only performing the first two phases, up until functional discovery; this is useful when non-functional requirements are not immediately available (e.g., they depend on Service Level Agreements that are pending). Such a framework can be of assistance to stakeholders in service engineering such as SBA designers or composition architects, by reducing the modeling effort required in order to create a composite service and also mitigating the effects of human error in the process.

### 5.2.1 Functional Composition and Verification

The first step in the proposed framework is to achieve functional composition and verification, given a repository of WSSL specifications and a set of functional goals, also expressed in WSSL. This is realized using WSSL planning and executed in the *ECL<sup>i</sup>PS<sup>e</sup>* constraint programming system. As defined in Section 5.1.4, for the WSSL plan to be effective and actually deliver complex composition plans and not merely sequential ones, heuristic encodings of the composition problem must

be supplied. There is a direct relation between the detail of the heuristic encoding and the effectiveness of the planner. The more effort is put by the designer in defining heuristic encodings, the more effective WSSL planning will be. In the typical case, we expect from the service composition designer to provide a rough skeleton of the composition process, with alternative choices with regard to actions and control constructs used, so that the planner does not go through all possible choices which would almost certainly lead to non-terminating executions.

It should also be taken into account that the framework expects more than one abstract plans to be produced in this step, otherwise the rest of the process may fail to deliver a satisfactory solution, due to not having alternative plans to fall back to if the single abstract plan never leads to a concrete process that satisfies all composition goals. Returning all possible planning problem solutions instead of only the first one found is expected to increase the time required to complete the task; a detailed evaluation of the whole framework, including the planning process, is provided in Chapter 7 and aims to shed light on scalability aspects, among others, in relation to real-world scenarios.

### **Verification**

Given a plan generated by the functional composition process, service verification aims to check that the composite service that corresponds to the plan meets some properties. The verification process in our framework focuses mainly on answering questions about the behavior of the composition and is conceptually identical to the verification tool presented in 5.1.3. Examples of the properties that can be verified are the following:

- *Composability* of a set of services: given a composition goal, the nature of the composition process yields results about whether this particular set can lead to a valid composition.
- *Liveness* and *safety* properties that check whether the composition plan realizes the goal behavior.

- *Conformance* of an observed composite service behavior to the corresponding plan specification and, in case conformance fails, derivation of possible *explanations* for the conflict in order to perform troubleshooting actions.

For example, a liveness property in our motivating scenario involves verifying whether the composition plan leads to the final report being delivered (either by mail or electronically), by proving  $Emailed(report) \vee Delivered(report)$ , where  $z$  is the final state, while a safety property would be to make sure that payment is performed for the correct payment form ( $Holds(HasInput(payment), z_{in}) \wedge \neg Holds(PayCompleted(payment2), z_{out})$ ). Verification queries of the third type can be answered due to WSSL's solution to the qualification problem. For instance, after executing the composite process in Fig. 2.1, we observe its behavior in the form of WSSL state descriptions and pose the query: in the final state  $z$ , is the goal  $holds(solved(status, location), z), holds(paycompleted(form), z), holds(hasoutput(report), z)$  satisfied? If the answer is no and no accidental qualifications have been expressed, then the observed behavior is deemed inconsistent with the composition specification. However, given the specification shown in Table 3.2, the framework deduces that an accident must have occurred, namely  $failure(deliv)$ . Such explanations are valuable for determining follow-up actions to unexpected situations, such as re-executing services that fail or adapting the composite process in order to replace them.

### 5.2.2 Specification-based Functional Discovery

For each task in the abstract plans produced by the functional planner, functional discovery is performed, which yields a set of *extended* plans, linking each task with (possibly more than one) concrete executable services. In our framework, we assume that all abstract tasks and concrete service candidates are associated with a WSSL specification. Hence, functional service matchmaking and selection relies on and is driven by WSSL specifications. Essentially, we need to check if there is any fluent that is included in the action precondition axiom of the

abstract task specification but is missing from the concrete service specification; the same must be done for state updates and causal relationships. The following definition formally states what is required in order to determine whether a candidate service specification matches an abstract task specification.

**Definition 5.2.1** Given two WSSL specifications R (abstract task) and T (concrete service), T matches R iff the following hold:

**Action precondition axioms** If  $Poss(R(x), z) \equiv \Pi_R(z)$  and  $Poss(T(x), z) \equiv \Pi_T(z)$ , then  $\Pi_T(z) \Rightarrow \Pi_R(z)$  must hold. With  $\Pi_R(z) \equiv Holds(f_{R1}, z) \wedge \dots \wedge Holds(f_{Rn}, z)$  and  $\Pi_T(z) \equiv Holds(f_{T1}, z) \wedge \dots \wedge Holds(f_{Tn}, z)$ , then  $\Pi_T(z) \Rightarrow \Pi_R(z) \equiv \{f_{Ri}, \dots, f_{Rn}\} \subseteq \{f_{Ti}, \dots, f_{Tn}\}$ .

**State update axioms** If  $Poss(R(x), s) \Rightarrow (\exists y_R)(\Delta_R(s) \wedge State(Do(R(x), s)) = State(s) + \theta_R^+ - \theta_R^-)$ ,  $Poss(T(x), s) \Rightarrow (\exists y_T)(\Delta_T(s) \wedge State(Do(T(x), s)) = State(s) + \theta_T^+ - \theta_T^-)$ , then  $\Delta_T(s) \Rightarrow \Delta_R(s)$  and  $State(Do(R(x), s)) = State(Do(T(x), s)) \equiv (\theta_R^+ \subseteq \theta_T^+) \wedge (\theta_R^- \subseteq \theta_T^-)$  must hold.

**State update axioms with ramifications** If  $Poss(R(x), s) \Rightarrow (\exists y_R)(\Delta_R(s) \wedge Ramify(z, \theta_R^+, \theta_R^-, z'))$  and  $Poss(T(x), s) \Rightarrow (\exists y_T)(\Delta_T(s) \wedge Ramify(z, \theta_T^+, \theta_T^-, z'))$  then  $\Delta_T(s) \Rightarrow \Delta_R(s)$  and  $\theta_T^+ \subseteq \theta_R^+ \wedge \theta_T^- \subseteq \theta_R^-$  must hold, in addition to the following equivalence of causal relationships:

**Causal relationships** If  $CR_R$  and  $CR_T$  are the two sets of causal relationships contained in R and T, respectively, then  $CR_R \subseteq CR_T$  must hold.

For instance, a service equivalent to *ReceivePay*, as defined in Tables 3.1 and 3.2, should have an action precondition axiom containing at least inputs *payform*, *location*, *creditCard* and precondition *Solved*, a state update that produces at least output *invoice* and postcondition *PayCompleted* and removes the *payform* fluent while the causal relationship that models card deactivation must also be included. Note that in Definition 5.2.1, we assume both specifications refer to the

same state variables  $z$  and  $z'$ . Also, this matchmaking process leads to either exact or super matches. It can be generalized to include subsumption and plug-in matches, if such relations are expressed between different fluents.

To realize functional matchmaking, we again rely on WSSL's implementation in FLUX and the *ECL<sup>i</sup>PS<sup>e</sup>* system (see Section 7.1.3 for a detailed evaluation of the functional discovery process in *ECL<sup>i</sup>PS<sup>e</sup>*). To determine whether the implication  $Holds(f_{Ti}, z) \subset Holds(f_{Ri}, z)$  is true, we state that `holds(f_Si, z)` are true (as Prolog facts) and check whether `holds(f_Ti, z)` can be derived. Alternatively, we can check whether `poss(Ti, Z), Z=[f_Ri]` succeeds. As far as state update axioms are concerned, we set a before state  $Z = [\theta_R^-]$  and an after state  $Z\_PR = [\theta_R^+]$  and check if `update(Z, T, Z_PR)` (or `ramify(Z, T, Z_PR)`) holds. There is no reason to check if  $CR_R \subseteq CR_T$  holds for the causal relationships, since if it does not, `ramify(Z, T, Z_PR)` will fail. Alternatively, we can check whether `state_update(Z, Ti, Z_PR), Z=[f_Ri], Z_PR=[Z-theta_minusR+theta_plusR]`. holds.

By employing specification-based discovery, we raise the problem to a higher level, disregarding any implementation details concerning the underlying services and relying only on the accompanying specifications. While this simplifies the discovery process, it can only be realized if the same specification language is used for all services, while specifications must refer to the same conceptual models. If either prerequisite does not hold, an alignment phase must precede matchmaking, based on translators from one specification language to another and adapters that modify a specification in order to refer to a different conceptual model. This holds for both functional and QoS parts of the specification.

### 5.2.3 Extended Plan Pruning and Ranking

Functional discovery results in a set of extended plans, whose number and size depends on two factors: the size of the subset of abstract plans returned by the planner that are actually realized (because at least a single implementation exists



for each task) and the number of concrete services available per task. While the latter depends primarily on the size of the service repository, the former depends on several aspects, such as functional goal complexity, the amount of candidate service specifications and the strictness of heuristic encodings employed by the planner.

An attempt at decreasing the size and complexity of the extended plan set is realized through the tasks of pruning and ranking. Pruning removes any concrete services included in extended plans that do not satisfy a local QoS goal. Goal and offering constraints should refer to the same attributes and use the same operators. An entire extended plan can be discarded if pruning removes all possible concrete services for a single task. For instance, in the running example, there is a constraint that requires the find mechanic task to complete in less than 60 seconds. Suppose that there exist two different specifications for this task, *FindMechSlow* and *FindMechFast*, but only *FindMechFast* is associated to concrete services that achieve the execution time goal. Given that knowledge, each extended plan that contains *FindMechSlow* must be discarded.

The plans that survive the pruning process are then ranked in order to determine the order in which they will be examined in the QoS-based selection phase that follows. Ranking is performed using three criteria. First, the maximum plan length is considered, i.e. the number of tasks included in the longest execution path in the plan. Second, the total number of tasks in the plan is taken into account. These first two criteria quantify plan complexity, so that plans achieving goals in less steps are preferred. Note that loops must be unfolded before calculating either of these metrics, using the maximum number of iterations.

Finally, ranking also relies on features that are domain/problem-dependent. For instance, plans that realize the payment process of the running example in a single task may be preferable, so that sensitive payment-related information is not accessed by more than one services. Such criteria are expected to be defined either by the composition requester or by designers with experience on the particular domain or problem. To check such problem-dependent criteria, each

extended plan must be parsed from start to end in order to determine whether the plan contains a task that goes against the defined criteria.

To produce a total rank score, the widely used Simple Additive Weighting method [Tzeng and Huang 2011] is employed. Weights are attributed uniformly, but expert knowledge can again come in handy here giving, for instance, more importance to problem-dependent criteria rather than those involving plan complexity. For evaluation purposes, we implemented a simple prototype pruning and ranking system in Java, that takes into account only problem-dependent features when calculating the rank score, while pruning assumes that values associated with local goals are already collected for all abstract tasks. The evaluation results are detailed in Section 7.1.5.

#### 5.2.4 QoS-based Selection

Following pruning and ranking, QoS-based selection is performed for the remaining plans, until one is found that satisfies all global QoS goals. Selection is performed based on the QoS profiles included in the WSSL specifications of the concrete services that remain for each task and each plan. In the best case scenario, only the top-ranked extended plan will need to be examined, hence reducing the overall time required for this phase. To realize this task, we rely on existing algorithms in literature and propose two different approaches.

If we have knowledge about execution path probabilities or have no issue assigning equal probabilities to each possible execution path, then we adopt the QoS-based selection algorithm that is defined in [Mello Ferreira et al. 2009], since it includes an aggregation process that relies on knowing such probabilities. Note that this algorithm also exploits utility functions that allow violation of global QoS goals in order to handle over-constrained QoS requirements; our framework currently does not support such soft constraints, hence the functions are modified to make sure all global goals are satisfied. Also, [Mello Ferreira et al. 2009] does not take into account all attribute categories that we have defined; In such

cases, the aggregation functions in Table 5.2 may be applied; this collection is essentially a simplification of Table 4.1, taking into account only sequence, which is the pattern used to model execution paths.

Table 5.2: Aggregation of QoS attributes per Execution Path

QoS Attribute Categories		Aggregation Function (per Execution Path)
Measurable	Temporal	$x_a = \sum_{i=1}^n x_i$
	Probabilistic	$x_a = \prod_{i=1}^n x_i$
	Cost	$x_a = \sum_{i=1}^n x_i$
	Reputation	$x_a = avg\{x_1, \dots, x_n\}$
	Throughput	$x_a = min\{x_1, \dots, x_n\}$
Unmeasurable	Boolean	$x_a = \begin{cases} false, & \exists i \cdot (x_i = false) \\ true, & otherwise \end{cases}$
	Ordered Set	$x_a = min\{x_1, \dots, x_n\}$
	Unordered Set	$x_a = \bigcap_{i=1}^n x_i$

On the other hand, if we do not consider execution path probabilities at all, we can perform aggregation based on the functions in Table 4.1, calculating values for all attributes related to global QoS goals, before using the QoS-based selection algorithms defined in [Kritikos and Plexousakis 2009a]. We implemented a prototype aggregator system in Java, based on processes that are modeled using BPMN [Object Management Group 2011], since it is the only process modeling language that supports all composition patterns we consider; BPEL is also another suitable candidate, provided that parallel execution is limited to the AND-AND and OR-OR cases. To implement BPMN-related functionality, the jBPM libraries [JBoss jBPM team 2013] were used. Algorithm 2 sketches the functionality of the aggregator system. An evaluation of the prototype is conducted in Section 7.1.6.

**Algorithm 2:** Aggregating QoS values in BPMN processes

**input** : A BPMN2 process (in XML), WSSL/XML documents for all participating services and the QoS attribute to be aggregated

**output**: The aggregated QoS value

**if** attribute is pattern-independent **then**

    nodeList  $\leftarrow$  GetNodes();

**foreach** node  $n$  in nodeList **do**

**if** node is a task **then**

            GetQoSValue();

**end**

**end**

*apply aggregation function on collected QoS values;*

**end**

**else**

    aggValue  $\leftarrow$  AggregatePath( $n$ );

**end**

---

AggregatePath( $n$ )

firstNode  $\leftarrow$  GetFirstNode( $n$ );

**if** firstNode is SplitNode **then**

    outC  $\leftarrow$  GetOutgoing();

**foreach** node  $c$  in outC **do**

        aggValue  $\leftarrow$  AggregatePath( $c$ );

**end**

**end**

**else**

    GetQoSValue();

**end**

*apply aggregation function on collected QoS values;*

### 5.2.5 Framework Use

WSSL/CVF is intended primarily for design time usage. We expect that SBA designers and composition architects can employ the framework in order to accelerate and facilitate the process of designing a composite SBA, given a specification of the goals requested by the service consumer and a specification repository, offering services from one or more providers.

The stakeholders associated with the exploitation and overall life-cycle of the framework include service providers, SBA designers and developers, composition architects, domain experts and service consumers/requesters. The typical use case that we envision involves service consumers/requesters expressing a complete specification of functional and non-functional requirements for the SBA they need. Service providers are then tasked with satisfying the request, by relying on SBA designers and composition architects that use WSSL/CVF.

In order for the framework to produce effective results, four prerequisites need to be satisfied:

1. Request specification should be expressed using WSSL, with the assistance of a service designer that has experience with the language.
2. Specifications for candidate services to participate in the composition process need to be expressed in WSSL as well. This is an indirect responsibility of service providers and can be fulfilled either by service developers or designers, before or after designing or implementing a service. Such specifications need to be kept up-to-date, reflecting changes resulting from service adaptation or service evolution processes.
3. Heuristic encodings need to be defined to guarantee termination and to make sure that control constructs other than sequence are taken into account. This is a responsibility of SBA designers and composition architects, who may also rely on experts on the particular domain/problem at hand.
4. Problem-dependent ranking criteria are also necessary to maximize opti-

mality. Once again, the know-how of SBA designers and composition architects, as well as domain/problem experts is invaluable.

### 5.2.6 Limitations

The choices of implementation languages and systems mentioned in this chapter come with a number of limitations that need to be taken into account. First of all, as already mentioned, implementing WSSL states in FLUX as Prolog lists means that states automatically become ordered lists, instead of unordered ones. This modification does not have any consequences on our framework since considering states as unordered lists is not required in any task included in the framework.

Prolog as a logic programming language has a series of limitations. The following cannot be expressed in Prolog:

1. Disjunctive and negative facts or conclusions
2. Quantification for facts and conclusions is limited: existential quantification is implicit for variables occurring only in the body of a rule while universal quantification is implicit for variables occurring in both the head and the body of a rule.
3. Second-order logic is not allowed directly.

In defining WSSL, we have taken into account only the first-order part of fluent calculus definitions; any second-order logic constructs were either simplified or were irrelevant to the service case. As far as the first three limitations are concerned, the rules that are defined in the FLUX kernels do not require such expressiveness; the same is true for clauses that encode a service domain.

The nature of FLUX planning also raises another interesting limitation. Suppose that we have a service  $A$  requiring a single input and a single precondition. If the fluents associated with the input and the precondition hold in a state  $Z$ , the planner will attempt to "execute" that service and the clause  $\text{poss}(A, Z)$ ,

`state_update(Z, A, Z1)` will succeed, resulting in state  $Z1$ . If the postconditions of the service do not remove either of the input and precondition fluents, the planner will repeat the process and the clause `poss(A, Z1), state_update(Z1, A, Z2)` will succeed again. As it becomes apparent, this scenario eventually results in a non-terminating execution. To address this, at least one of the associated inputs or preconditions of a service must behave like a token that is consumed and removed after a successful execution.

A related limitation involves planning using incomplete states in the form of  $Z0 = [F1, \dots, FN \mid Z]$ , associated with a set of constraints on  $Z$ . Searching for a solution to a planning problem starting from an incomplete initial state, the planner will assume, at any given state, that the planning goal has been achieved, provided that this does not violate the constraints. In cases where no heuristic encoding is used, or when the heuristic encoding is not restrictive enough, attempting to find all possible solutions will quickly result in non-terminating executions; this is due to the fact that the planner can assume that the goal has been reached, either by not executing any action, or by executing any possible combination of actions, since it can always assume that the required goal is achieved by the incomplete (unknown) part of the current state.

Another limitation concerns the Java-*ECL<sup>i</sup>PS<sup>e</sup>* interface. When communicating goals and results through the interface, all variables are replaced by the null token. In the case of results, this poses a presentation problem when attempting to deliver the result to the user through the interface of the WSSL verification tool. This problem is circumvented by keeping a table of variable names and using it to translate the returned result.

Finally, the jBPM libraries currently do not support activity looping, since there is no way to access loop characteristics of a node. In order to support looped activities in aggregation, we adopt a naming convention: names of looped activities are prefixed with a sequence that starts with "Loop\_", followed by the number of iterations and ending with an underscore, e.g., "Loop\_10\_TaskName" represents a looped execution of TaskName for a maximum of 10 iterations.





# Chapter 6

## Property Analysis

### Contents

---

<b>6.1 Correctness</b>	<b>148</b>
6.1.1 Holds	148
6.1.2 Minus, Plus and Update	149
6.1.3 Causes and Ramify	150
6.1.4 Control Flow	150
6.1.5 Complete FLUX Kernel	150
<b>6.2 Decidability and Complexity</b>	<b>151</b>
6.2.1 Core WSSL	151
6.2.2 WSSL Extensions	153
<b>6.3 Applicability and Practical Concerns</b>	<b>154</b>
6.3.1 WSSL Grounding to WSDL	155
6.3.2 Generating WSSL Specifications	160
6.3.3 Generating BPMN from WSSL plans	164
6.3.4 Integrating WSSL into USDL	165
<b>6.4 Conclusions</b>	<b>165</b>

---

In this chapter, a series of interesting properties in relation to WSSL is analyzed, namely correctness of the associated FLUX kernels, decidability and com-

plexity of the underlying fluent calculus theory, as well as applicability of the language in terms of its connection to other related or complementary languages in service science. The discussion intends to reinforce the belief that WSSL is a language of high usability despite the fact that, at first glance, its high expressivity may indicate otherwise.

## 6.1 Correctness

WSSL/CVF, analyzed in Chapter 5, relies on two slightly modified FLUX kernels in order to implement the fluent calculus foundations of WSSL. In this section, we investigate correctness for these kernels with regard to their WSSL foundations, based on the correctness results of the initial FLUX kernel, as reported in [Thielscher 2005b]. Proving correctness aims at supporting the use of FLUX in safety-critical service composition and verification tasks.

### 6.1.1 Holds

Beginning our investigation with clauses that constitute the Basic FLUX kernel, Theorem 2.2 in Chapter 2 of [Thielscher 2005b] proves correctness for *Holds* through the following:

$COMP[P_{kernel}] \models Holds(f, \llbracket z \rrbracket) \text{ iff } \Sigma_{state} \models Holds(f, z) \text{ and}$

$COMP[P_{kernel}] \models \neg Holds(f, \llbracket z \rrbracket) \text{ iff } \Sigma_{state} \models \neg Holds(f, z) \text{ where } P_{kernel} \text{ is}$   
a FLUX kernel,  $COMP[P_{kernel}]$  is its completion, giving its declarative semantics,  $\Sigma_{state}$  is a fluent calculus state signature,  $z$  is a ground fluent calculus state (consisting of a finite set of fluents without variables) and  $\llbracket z \rrbracket$  is a ground FLUX state containing the same fluents as  $z$ . Since the proof makes no claim for  $f$ , other than it being ground, then the proof also holds when  $f = HasInput$  or  $f = HasOutput$ .

### 6.1.2 Minus, Plus and Update

Lemmas 2.4, 2.5 and Theorem 2.6 in [Thielscher 2005b] prove correctness for  $Minus(z, \theta^-, z')$ ,  $Plus(z, \theta^+, z')$  and  $Update(z, \theta^+, \theta^-, z')$ , respectively. Since our kernel includes a modified definition of  $Plus$ , we need to make sure that the proof of correctness still holds.

**Lemma 6.1.1** Consider the basic FLUX kernel  $P_{kernel}$  and the corresponding fluent calculus signature. Let  $z, \theta^+, z'$  be ground states; then for any  $\llbracket z \rrbracket$  and  $\llbracket \theta^+ \rrbracket$ , there exists  $\llbracket z' \rrbracket$  such that  $COMP[P_{kernel}] \models Plus(\llbracket z \rrbracket, \llbracket \theta^+ \rrbracket, \llbracket z' \rrbracket)$  iff  $\Sigma_{state} \models z' = z \circ \theta^+$ .

**Proof**  $COMP[P_{kernel}]$  entails the definition

$$\begin{aligned}
Plus(z, \theta^+, z') \equiv & (\theta^+ = [] \wedge z' = z) \vee \\
& (\exists f, \theta_1^+, z_1, z_2, v) (\theta^+ = [f | \theta_1^+] \wedge \\
& \neg Holds(f, z) \wedge z_1 = [f | z] \wedge \\
& (f = HasOutput(v) \wedge z_2 = [HasInput(v) | z_1] \vee \\
& f \neq HasOutput(v) \wedge z_2 = z_1) \vee \\
& Holds(f, z) \wedge z_2 = z \wedge Plus(z_2, \theta_1^+, z')) \quad (6.1)
\end{aligned}$$

Let  $\llbracket \theta_1^+ \rrbracket = [f_1, \dots, f_n]$ . We prove the claim by induction on  $n$ . If  $n = 0$ , then  $\llbracket \theta_1^+ \rrbracket = []$  and  $Plus(\llbracket z \rrbracket, [], z') \equiv z' = \llbracket z \rrbracket$ . Also  $\theta_1^+ = \emptyset$  and  $\Sigma_{state} \models z' = z \circ \theta_1^+ = z \circ \emptyset = z$ , which proves the claim.

If  $n = 1$  and  $f_1 = HasOutput(v)$  then from (6.1) we have  $Plus(\llbracket z \rrbracket, HasOutput(v), z') \equiv (\exists z_1, z_2, v) (\neg Holds(f, \llbracket z \rrbracket) \wedge z_1 = [HasOutput(v) | \llbracket z \rrbracket] \wedge z_2 = [HasInput(v) | z_1] \wedge Plus(z_2, [], z'))$  which proves the claim according to the proof of correctness of  $Holds(f, z)$  and the induction hypothesis.

If  $n = 1$  and  $f_1 \neq HasOutput(v)$  then from (6.1) we have  $Plus(\llbracket z \rrbracket, f_1, z') \equiv (\exists z_1, z_2, v) (\neg Holds(f, \llbracket z \rrbracket) \wedge z_1 = [f_1 | \llbracket z \rrbracket] \wedge z_2 = z_1 \wedge$

$Plus(z_2, [], z')$  which again proves the claim according to the proof of correctness of  $Holds(f, z)$  and the induction hypothesis.

The proof is achieved in the same way in case  $n > 0$  and for any combination of fluents ( $HasOutput$  or other).

### 6.1.3 Causes and Ramify

The introduction of *Causes* and *Ramify* in FLUX in order to translate the fluent calculus solution to the ramification problem affects correctness of the update process, since it requires that the application of a finite set of causal relationships always reaches a final state; in any other case, total correctness is not guaranteed. Hence, care must be taken in the encoding of causal relationships, so that no application of them leads to an infinite cycle. In the case of service specifications, however, as analyzed in Section 3.2.3, only direct causal relationships between condition pairs are expected to be of interest, hence the probability of endangering correctness due to arbitrary chains of causal relationships is insignificant.

### 6.1.4 Control Flow

Correctness is not affected by the clauses that handle control flow for service composition, since they consist of clauses, the correctness of which is already proven: *Holds* clauses, *Poss* clauses that consist of *Holds* clauses and *StateUpdate* clauses which consist of *Update* or *Ramify* clauses.

### 6.1.5 Complete FLUX Kernel

The complete FLUX kernel introduces several modifications, following those introduced in the specification of General FLUX in [Thielscher 2005b]. Hence, any existing correctness results for the newly added clauses are adopted as-is. Correctness of the constraint solver is proven in Chapter 4.3 of [Thielscher 2005b], while Lemmas 4.9, 4.10 and 4.11 and Theorem 4.12 in [Thielscher 2005b] handle

correctness for incomplete state updates. It is worth mentioning that update is proven to be sound but incomplete, since in some cases FLUX infers a weaker state specification than the one derived from the fluent calculus foundations.

## 6.2 Decidability and Complexity

WSSL and its extensions were designed with high expressivity as a fundamental goal in order to be able to support solutions to the frame, ramification and qualification problems. However, the tradeoff between expressivity and decidability means that the more expressive a language gets, the more possible it is for the decision problem to be unsolvable. Since WSSL is founded on the fluent calculus, we need to review existing decidability results for the fluent calculus, as well as results for the complexity of the decision problem and determine how these results adapt to WSSL.

### 6.2.1 Core WSSL

[Lehmann and Leuschel 2000] proves that entailment for the full propositional fluent calculus is undecidable. Entailment becomes decidable for the fragment that restricts formulas to conjunctions of two *Holds* formulas. Independently, [Hölldobler and Kuske 2000] investigates decidability boundaries by making a distinction between a fluent calculus formalism that supports resources and one that only supports properties. The former requires that states are represented by multisets of fluents, possibly containing more than one copies of the same fluent. The latter restricts state representations to sets of fluents, where each fluent can appear at most once. Note that in the case of service specifications, this restriction is in line with the concept of service behavior, since we are only interested in fluents as properties that may or may not hold.

[Hölldobler and Kuske 2000]'s investigation first focuses on a monadic second-order fragment of the fluent calculus with resources: with the exception of situ-

ations, all other variables participate only in first-order formulas, arbitrary function symbols and predicates are not allowed and fluents are represented by constants. Entailment is decidable only for *monadic* queries (i.e. formulas without free variables), for *restricted* state update axioms (i.e. ones that allow only boolean combinations of formulas of the form  $\phi(\text{State}(s))$  and  $\phi(\text{State}(\text{Do}_a(s)))$ , where  $\phi$  is a state formula with one free variable). However, it is not elementary decidable, since complexity for the monadic decision problem cannot be described by a function using addition, multiplication and exponentiation. On the other hand, entailment for the fluent calculus with properties is also decidable for monadic queries but for the more expressive case of unrestricted state updated axioms. Entailment becomes undecidable again if two unary function symbols are introduced, mapping fluents to fluents.

Following a different path towards assessing decidability for the fluent calculus, [Schiffel and Thielscher 2006] presents a complete and correct bidirectional translation process between the situation calculus and the fluent calculus. This translation process guarantees equivalence of the entailment procedure for the two calculi. Hence, it can be argued that any decidability results obtained in research on the situation calculus can be applied for the fluent calculus as well.

One case of decidability results in the situation calculus is presented in [Gu and Soutchanski 2007] and is of particular interest due to its direct relation to service description. The authors restrict situation calculus to a two-variable fragment with counting,  $C^2$ , which has been proven to be decidable with NEXPTIME complexity [Pacholski et al. 2000] and also add Description Logic capabilities, such as the expression of TBox and ABox statements. The authors examine the expressivity of such an action language for the case of service domains, pointing out the fact that if both atomic services and properties affected by them can be expressed using only two parameters, then they can be expressed using  $C^2$ , with decidable entailment. Additionally, [Milicic 2008] proves that the  $C^2$  fragment of the situation calculus is more expressive than any of the DL fragments considered in literature in relation to action formalisms.

Based on these results, we can deduce that entailment is decidable for the fluent calculus fragment that is equivalent to the  $C^2$  fragment of the situation calculus, obtained using the translation process defined in [Schiffel and Thielscher 2006]. This fragment allows for descriptions of services represented by actions with at most 2 arguments and fluents with at most 3 arguments, one of which is a situation variable. Moreover, this fragment is considerably more expressive than the ones investigated by [Lehmann and Leuschel 2000] and [Hölldobler and Kuske 2000] and, based on the results of [Milicic 2008], is the most expressive one out of all logic-based action formalisms that are not based on narrative and have been considered for service specification.

### 6.2.2 WSSL Extensions

While decidability has been investigated in literature for the core fluent calculus, no decidability results exist for the extensions that cover the ramification and qualification problems, and incomplete and knowledge states. As far as the solution to the ramification problem is concerned, it is obvious that the *Causes* predicate cannot be defined in the aforementioned decidable fragment of the fluent calculus. Hence, the only way ramifications can be modeled in a decidable way is if they are degenerated into direct effects of a state update.

The solution to the qualification problem requires the inclusion of accident-based clauses in both action precondition and state update axioms as well as the introduction of a default theory in order to assume away accidents. Decidability is not affected by accident-based clauses, provided that they conform to the two-variable restriction, which is achievable at least for expressing the no-accident case (which follows the form  $(\forall c)\neg Acc(c, s)$ ).

Default theories are shown to be decidable in [Besnard et al. 1983], provided that they consist exclusively of *free-defaults* (i.e. defaults without prerequisites) and consequences and axioms in the defaults are *predefinite variable clauses* (i.e. they are function-free and all variables occurring in a positive literal also occur in

a negative one). The default theory introduced in order to assume away accidents ( $\Delta = (\{\frac{\neg Acc(c,s)}{\neg Acc(c,s)}\})$ ) satisfies these requirements, hence decidability results are not affected by it.

Moving on to partial observability, it should be noted that the decidable fragment of the fluent calculus can only express incomplete or knowledge states that are defined using constraints that conform to its restrictions. Thus, state formulas that are associated with these definitions have to belong at most to the fragment of the fluent calculus that is equivalent to  $C^2$ ; in other words, no more than two variables are allowed and the signature must contain only predicate symbols.

### 6.3 Applicability and Practical Concerns

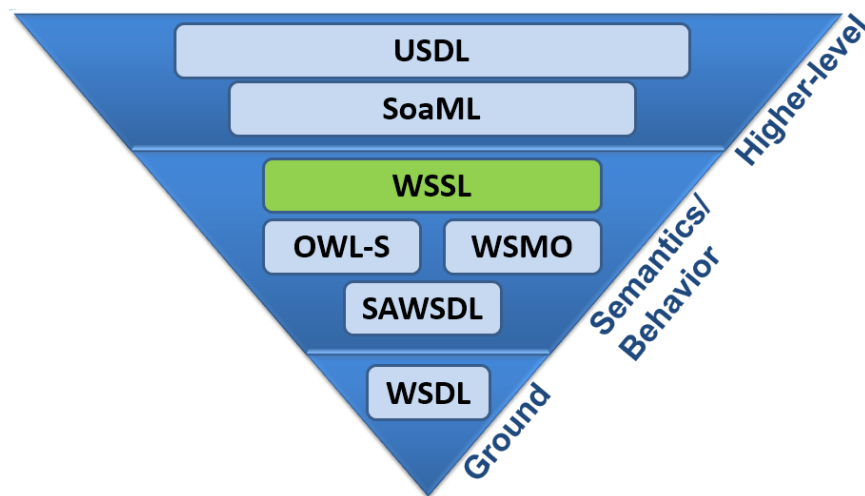


Figure 6.1: Placement of WSSL with regard to current and former service description efforts

WSSL is designed as a language at the level of OWL-S [Martin et al. 2004] and WSML [WSML Working Group 2008a], focusing on representation completeness with regard to the frame, ramification and qualification problems. As illustrated in Fig. 6.1, this characteristic places WSSL at a position below higher-level languages such as SoaML [Object Management Group 2012b] or the more recent



USDL [Kadner et al. 2011, Oberle et al. 2013], but above ground languages such as WSDL [Chinnici et al. 2007]. This section makes a case for the applicability of WSSL by concretizing its relation with most of the aforementioned service description languages.

### 6.3.1 WSSL Grounding to WSDL

WSSL specifications provide a complete behavioral description of a service in terms of IOPEs, effectively answering the question of what a particular service does, or how it affects the state of the world. A behavioral description does not concern itself with information about how to invoke a service or the format of messages exchanged. WSDL<sup>1</sup> is considered the current industry standard for providing such information, enabling a user to invoke a service and exchange information, regardless of service implementation. Hence, it is important to establish a connection between WSSL and WSDL, in order to bring WSSL closer to real-world scenarios and maximize its usability. This connection is defined in terms of a *grounding* mechanism, that maps a WSSL specification to an existing service described in WSDL, following the equivalent mechanism employed by WSML.

Table 6.1 shows a basic correspondence between parts of a WSSL specification that are relevant to service grounding and parts of a WSDL description. Note that this correspondence assumes a similar level of granularity between the two descriptions. However, there is also the possibility that one of them is more coarse-grained (or more fine-grained) than the other. For instance, a WSSL service specification may match a set of operations instead of a single one, or a single WSDL operation may correspond to more than one WSSL specifications. Similarly, a set of WSSL inputs or outputs may correspond to a single input or output message and vice-versa.

Each of the WSSL elements included in Table 6.1 is associated with a *grounding* property that takes an IRI as value, mapping the particular element to the corre-

---

<sup>1</sup>A concise description of WSDL 2.0 can be found in Section 2.3.1

Table 6.1: Mapping between WSDL and WSSL

WSSL	WSDL
Service name	Operation name
Input	Input message InFault message
Output	Output message OutFault message

sponding one in a WSDL description. These IRIs follow the format that is defined in Appendices A.2 and C of [Chinnici et al. 2007] and are created by combining the namespace URI with fragment identifiers that unambiguously identify components on any level of granularity within a WSDL file. Note that in case we have access to a semantically annotated WSDL description (expressed using SAWSDL, summarized in Section 2.3.2), then we can use the `modelReference` attributes of each input and output as names for the corresponding WSSL elements.

### Grounding Example

Suppose that there is a WSDL description of a service that implements the *Login* task of the running example. Part of this description is shown in Listing 6.1:

```

1 <binding>
2   name="SOAPLoginService"
3   interface="LoginInterface"
4   type="http://www.w3.org/2004/08/wsdl/soap12"
5   wssoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/"
6 </binding>
7
8 <service
9   name="LoginService"
10  interface="LoginInterface">
11  <endpoint
12    name="normal"

```

```
13     binding="SOAPLoginService"
14     address="http://www.example.org/login"/>
15 </service>
16
17 <interface name="LoginInterface">
18     <operation
19         name="login"
20         pattern="http://www.w3.org/ns/wsd1/in2-out">
21
22         <input element="ns:loginForm"/>
23         <input element="ns:user"/>
24         <output element="ns:loginConf"/>
25     </operation>
26 </interface>
```

Listing 6.1: WSDL description of Login service

Note that we assume that *in2-out* is a predefined message exchange pattern that expects 2 input messages, labeled *In1* and *In2* and produces 1 output message, labeled *Out*. Part of a WSSL specification for the *Login* task that is grounded to the WSDL description above would be expressed as shown in Listing 6.2 below:

```
1 <service
2     name="wssl#Login"
3     grounding="http://example.org/Login.wsd1#
4     wsd1.interfaceOperation(LoginInterface/login)"/>
5
6 <input
7     name="wssl#payForm"
8     grounding="http://example.org/Login.wsd1#
9     wsd1.interfaceMessageReference(LoginInterface/login/In1)"/>
10
11 <input
12     name="wssl#user"
13     grounding="http://example.org/Login.wsd1#
14     wsd1.interfaceMessageReference(LoginInterface/login/In2)"/>
15
16 <output
```

```
16     name="wssl#payFConf"
17     grounding="http://example.org/Login.wsdl#
18     wsdl.interfaceMessageReference(LoginInterface/login/Out)"/>
19 </service>
```

Listing 6.2: WSSL specification of Login service

### Generating WSDL from WSSL

The discussion so far assumes that WSSL specifications are created as an enhanced and complete description of already existing services that are associated with WSDL descriptions and that the grounding mechanism associates specifications with WSDL endpoints. However, one can envision a different engineering scenario where services are first designed at the specification level using WSSL, and their implementation is performed separately based on such specifications. In this scenario, a default WSDL description should be automatically generated based on a WSSL specification, so that the associated service has the ability to receive and emit messages. The WSDL document is generated based on the template shown in Listing 6.3:

```
1 <description xmlns="http://www.w3.org/2005/05/wsdl"
2     targetNamespace="namespace ID"
3     xmlns:xs="http://www.w3.org/2001/XMLSchema"
4     xmlns:wsoap="http://www.w3.org/2005/05/wsdl/soap"
5     xmlns:wssl="http://www.example.org/wssl">
6
7     <types>
8     Types declaration
9     </types>
10
11     <interface name="interface name">
12         <operation
13             name="service name"
14             pattern="pattern name"
15
```

```
16         <input element="input name"/>
17         ...
18         <output element="output name"/>
19     </operation>
20 </interface>
21
22 <binding name="DefaultSOAPBinding" >
23     type="http://www.w3.org/2004/08/wsdl/soap12"
24     wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP/" >
25 </binding>
26
27 <service name="service name"
28     interface="interface name">
29     <endpoint name="endpoint name"
30         binding="DefaultSOAPBinding"
31         address="endpoint address"/>
32 </service>
33 </description>
```

Listing 6.3: WSDL template for WSSL-to-WSDL generation

The placeholders are filled in as follows:

**namespace ID:** The namespace, as defined in the WSSL document.

**Types declaration:** For each ontology concept used in the WSSL document, one XML Schema type is defined.

**service name:** The value of the name attribute of the service element in the WSSL document.

**interface name:** The service name, followed by the string "Interface".

**endpoint name:** The service name, followed by the string "Endpoint".

**pattern name:** The name of a previously defined message exchange pattern matching the particular service (e.g., in-out).

**input name:** The value of the name attribute of an input element in the WSSL document.

**output name:** The value of the name attribute of an output element in the WSSL document.

**endpoint address:** The value of the endpoint attribute of the service element in the WSSL document. If it is empty, then an endpoint must be externally specified for the service to be accessible.

Note that the choice of a message exchange pattern depends primarily on the number of inputs and outputs included in the WSSL specification. The automatically generated WSDL document should serve as a default starting point for designers or developers that are tasked to create a service based on a WSSL specification.

### 6.3.2 Generating WSSL Specifications

Apart from associating WSSL specifications with existing WSDL descriptions (or generating such descriptions), an additional way of extending applicability of the language is to port existing service descriptions to WSSL. In order for a language to qualify for this process, it should at least offer the ability to express IOPEs, since WSSL specifications rely on such information. The languages that we consider are WSML [WSML Working Group 2008a] and OWL-S [Martin et al. 2004]. Note that in both of these cases, service descriptions are associated with an ontology. Since WSSL is not an ontology language, any concept included in the resulting specification is linked to the ontologies defined in the initial descriptions. Therefore, parts of the initial WSML and OWL-S documents are eventually re-used as service ontologies for the resulting WSSL specifications.

**From WSMML**

The translation process is based on the abstract syntax definition of WSMML<sup>2</sup>. The subset of this syntax that is relevant to WSSL is shown below:

```

<wsml> ::= <wsmlvariant>? <namespace>? <definition>*

<definition> ::= <goal>
| <ontology>
| <webservice>
| <mediator>
| <capability>
| <interface>

<webservice> ::= 'webService' <iri>? <header>* <nfp>* <capability>? <interface>*

<capability> ::= 'capability' <iri>? <header>* <nfp>* <pre_post_ass_or_eff>*

<pre_post_ass_or_eff> ::= 'precondition' <axiomdefinition>
| 'postcondition' <axiomdefinition>
| 'assumption' <axiomdefinition>
| 'effect' <axiomdefinition>

<axiomdefinition> ::= <id>? <annotations>?
| <id>? <annotations>? <log_definition>

<log_definition> ::= 'definedBy' <log_expr>+

<interface> ::= 'interface' <iri>? <header>* <nfp>* <choreography>? <orchestration>?

<orchestration> ::= 'orchestration' <iri>?

<nfp> ::= 'nfp' <attributevaluenfp> <log_definition_nfp>?
| 'nonFunctionalProperty' <attributevaluenfp> <log_definition_nfp>?

<attributevaluenfp> ::= id 'hasValue' <valuelistnfp> <annotations>?

```

<sup>2</sup>For a condensed description of WSMML and WSMO, please refer to Section 2.3.4

$$\begin{aligned}
\langle \text{valuenfp} \rangle &::= \langle \text{basevalue} \rangle \\
&| \langle \text{variable} \rangle \\
\langle \text{valuelistnfp} \rangle &::= \langle \text{valuenfp} \rangle \\
&| \text{'\{'} \langle \text{valuenfp} \rangle \langle \text{morenfpvalues} \rangle^* \text{'\}' } \\
\langle \text{morenfpvalues} \rangle &::= \text{'\,'} \langle \text{valuenfp} \rangle \\
\langle \text{log\_definition\_nfp} \rangle &::= \text{'definedby'} \langle \text{log\_expr} \rangle^+
\end{aligned}$$

Based on this syntax, the generation of a WSSL specification from a WSML description is realized according to the following steps:

1. The `<iri>` associated with the `<webservice>` definition is used as a service identifier
2. WSSL inputs and outputs are derived from the `<orchestration>` definition. Note that no orchestration syntax has been specified by the WSML group, hence the derivation process cannot be specified in more detail.
3. WSSL preconditions are formed based on the precondition and assumption axioms included in the `<capability>` definition, while WSSL postconditions are based on the postcondition and effect axioms.

Axiom definitions for preconditions, postconditions, assumptions and effects in WSML employ a set of logical connectives that is almost equivalent to the one used in WSSL logical expressions. Most operators are shared between the two languages ( $=, \neq, \wedge, \vee, \neg, \forall, \exists, \supset, \equiv$ ), with the single exception of the WSML connective *naf*, which represents negation as failure. Since WSSL employs only classical negation, any occurrence of *naf* is replaced by  $\neg$ . WSML axioms may also include auxiliary symbols that represent ontological concepts such as *memberOf*, *subConceptOf*, *ofType* and *impliesType*. Such axioms do not belong in a WSSL specification and cannot be ported, since WSSL does not include any ontology definitions.



Table 6.2: Mapping between OWL-S and WSSL

OWL-S Service Profile	WSSL
Service name	serviceName property
Input	Input instances (hasInput property)
Output	Output instances (hasOutput property)
Precondition	Precondition instances (hasPrecondition property)
Postcondition	Result instances (hasResult property)

### From OWL-S

Generating WSSL specifications from OWL-S<sup>3</sup> ontologies is more straightforward than the case of WSML, since the ServiceProfile class of OWL-S contains all the relevant information for specifying service behavior. Table 6.2 presents a mapping between OWL-S properties and WSSL concepts. OWL-S does not define a specific logic language for expressing preconditions and postconditions, although several ones are mentioned as possible candidates, including SWRL. Thus, any translation mechanism from OWL-S to WSSL depends on the choice of logic language in OWL-S. Note that OWL-S Results are essentially coupled outputs and postconditions (effects), hence only the conditional part of them is ported to WSSL.

Since OWL-S and WSML do not take into account the ramification and qualification problems, the resulting WSSL specifications cannot be considered complete. For instance, there may be some ramifications that result from service postconditions, while one may wish to include accidental cases in postconditions. Such additional information can be added to the initially derived specifications by the service designer or the service provider.

<sup>3</sup>For an overview of OWL-S, please refer to Section 2.3.3

Table 6.3: Mapping between BPMN and WSSL for composition

WSSL	BPMN
service	Service Task
; (sequence)	Sequence Flow
· (AND-Split/AND-Join)	Parallel Gateway
+ (OR-Split/OR-Join)	Inclusive Gateway
$\oplus$ (XOR-Split/XOR-Join)	Exclusive Gateway
<i>If</i> (conditional)	Exclusive Gateway
<i>Loop</i> (iteration)	Activity Looping

### 6.3.3 Generating BPMN from WSSL plans

The WSSL composition framework analyzed in Chapter 5.2 produces textual descriptions of composite services that achieve the planning goals. In order for such plans to become executable, two important tasks must be carried out: service grounding and business process generation. The first task can be realized in a straightforward way, based on the discussion earlier in this section. Generating a composite business process requires a predefined mapping between composite constructs supported by WSSL planning and the ones supported by the business process modeling language employed. For our purposes, we choose BPMN 2.0 [Object Management Group 2011] since it is the only business process modeling language that supports all WSSL composite constructs. Table 6.3 shows the correspondence between the two sets of control constructs.

Note that exclusive gateways are used for both XOR-Split/XOR-Join and conditional execution. The difference lies in the ConditionExpression that is associated with each Sequence Flow that is involved with the gateway: in the case of conditionals, the expression is explicitly associated with the truth value of the if-condition. The creation of the composite process can either be done manually by the service designer, or semi-automatically, by parsing the textual plan specification produced by the WSSL composition framework and applying the one-to-one

Table 6.4: WSSL/USDL Integration

WSSL	USDL
Service	serviceName property
Input formula	Interface (Technical module)
Output formula	Interface (Technical module)
Precondition Axiom	Function - precondition association
State Update Axiom	Function - postcondition association
Causal Relationships	N/A
Default Theory	N/A

mapping rules of Table 6.3.

#### 6.3.4 Integrating WSSL into USDL

USDL [Kadner et al. 2011] is designed with the requirements of conceptualization and modularity in mind, while care has been taken so that individual modules are not bound to existing service description efforts, to the extent possible. Thus, from a theoretical viewpoint, WSSL specifications can be used as a basis for the Functional and Technical modules of USDL. Table 6.4 offers a possible connection between WSSL and concepts in the UML class diagram of the USDL Functional and Technical modules that are included in [Oberle et al. 2013]. Note that there is no direct equivalent for causal relationships or default theories. Including these aspects would require a modification of the Functional module class diagram. Also note that in a similar manner, WSSL QoS profiles can be integrated in the USDL Service Level module.

## 6.4 Conclusions

Through the analysis in this chapter, two significant conclusions can be drawn. First, the fluent calculus theory on which WSSL relies is, in general, undecidable.

It can be made decidable and even elementary decidable, at the cost of severely limiting the ability to provide specifications for a wide range of service behaviors. However, as it is clarified in the following chapter, undecidability does not affect the effectiveness and efficiency of WSSL/CVF, hence it does not hinder the usability of the language in real-world scenarios.

The applicability discussion in the second part of the chapter supports the conclusion that WSSL can be employed in practical applications, in cases where highly expressive specifications of service behavior are required. The grounding mechanisms and translation schemas that were presented pave the way for the realization of a WSSL-based design framework on top of the existing composition and verification framework that can assist a service designer in creating and annotating specifications for existing or newly-implemented services, in addition to designing, verifying and executing composite service processes.

# Chapter 7

## Evaluation

### Contents

---

<b>7.1</b>	<b>Evaluation of individual components . . . . .</b>	<b>169</b>
7.1.1	WSSL-to-FLUX Translation . . . . .	169
7.1.2	WSSL Planning . . . . .	170
7.1.3	Specification-based Functional Discovery . . . . .	179
7.1.4	Extended Plan Pruning . . . . .	182
7.1.5	Extended Plan Ranking . . . . .	184
7.1.6	QoS Aggregation . . . . .	187
7.1.7	Conclusion . . . . .	187
<b>7.2</b>	<b>Overall Evaluation . . . . .</b>	<b>190</b>

---

This chapter is devoted to a thorough evaluation of the various components of WSSL/CVF, the framework that was analyzed in Chapter 5. Two separate sets of experiments were conducted and are presented in the following sections. The first set examines each component of the framework individually and attempts to investigate the effect of all parameters that are of interest in each case. The vast majority of these experiments focuses on evaluating performance scalability, in terms of computation time, by varying specific parameters; the only exception is the experiment that concerns the extended plan ranking process, which evaluates optimality. The second set of experiments is essentially an overall evaluation of

the complete framework, again in terms of computation time, for 5 increasingly complex composition problems; in this set, a few parameters vary while the rest stay fixed based on assumptions that ground the problems in real-world scenarios.

Given the fact that the majority of available concrete services as well as service test sets contain services described using WSDL, they are unsuitable for our evaluation. On the other hand, the few test sets that contain Semantic Web services are too simplistic for the purposes of our evaluation; for instance, in OWL-S TC [Klusch et al. 2010] only 186 out of 1083 services have preconditions and postconditions (in most cases only one of each), with all other service descriptions containing only inputs and outputs and ramifications are obviously not included. Consequently, we generate synthetic WSSL specifications of different sizes, depending on the needs of each experiment.

The complexity of the generated specifications matches or even surpasses that of real service data. More concretely, specifications contain 1 to 5 IOPE quads, along with 1-3 ramifications; in contrast, actual service specifications, such as the ones contained in OWL-S TC [Klusch et al. 2010], have multiple inputs and outputs (up to 10 in total), but have at most 1-2 preconditions and effects while they contain no ramifications. Experiments involve repositories that contain up to 500 different specifications, i.e. up to 500 distinct functionality sets. Given the fact that each of these specifications may correspond to multiple implementations, the complexity is equivalent to actual service repositories containing a few thousands of concrete services. Concerning plan complexity, we evaluate the most complex cases where 50% or 100% of all specifications in the repository are required to achieve the plan goal. Finally, QoS-based evaluation assumes that each implementation is associated with 3 different quality profiles; actual service data almost never contain more than 3 QoS profiles.

The generated specifications contain IOPEs with generic names, e.g., inputs and outputs are named with the prefix "io", followed by a number, while preconditions and postconditions have the prefix "cond", again followed by a number. The

evaluation was performed on a Windows® 8 system with an Intel® Core™ i7-740QM processor running at 1.73GHz, with 6 GB RAM. The computation time values are an average of 20 runs.

## 7.1 Evaluation of individual components

The evaluation process first focuses on the individual modules that were implemented and analyzed in Chapter 5. Apart from evaluating computation time scalability for each separate module, experiments are also carried out to investigate optimality of the extended plan ranking mechanism. The rest of this section presents these experiments organized by task, beginning with the WSSL-to-FLUX translation process, followed by the individual tasks of WSSL/CVF that are implemented as part of this thesis, namely functional composition through WSSL planning, specification-based functional discovery, extended plan pruning and ranking and QoS aggregation. Note that the cost of the verification process is trivial since it essentially involves checking whether one or more fluents hold in a state, hence there is no need to perform any verification-specific evaluation.

### 7.1.1 WSSL-to-FLUX Translation

Translating a WSSL/XML document to a FLUX program is a mandatory step before attempting any process, whether it is a simple service verification process, as defined in Section 5.1.3, or a complete composition process using WSSL/CVF. Hence, it is important to evaluate the overhead of the translation process in terms of runtime and memory consumption and investigate whether scalability is achieved, as specifications become more complex. Synthetic WSSL specifications are created, containing from 1 to 500 pairs of preconditions and postconditions using randomly generated, uniquely named fluents. Note that we choose an unnaturally high number of pairs as an upper limit, in order to better understand the behavior of the translation process, even for cases that are beyond real-world

scenarios. The runtime values are an average of 20 runs, while memory consumption is presented before and after garbage collection.

As we can see in Figs. 7.1 and 7.2, runtime stays less than 600ms even for specifications containing 500 pairs of preconditions and postconditions, and actual memory consumption peaks at around 8000KB. Based on these results and the fact that actual service specifications contain at most tens of pre/post pairs (even in the case of service composition), we can safely assume that the overhead posed by the translation process from WSSL to FLUX is insignificant.

### 7.1.2 WSSL Planning

In order to evaluate functional composition via planning with WSSL, we run a series of experiments, calculating the time needed for the planner to produce a valid service composition plan, given a set of services, an initial state and a goal state. The parameters that are of interest in this evaluation are the following:

1. Size of the specification repository. Note that we assume that different implementations of the same functionality are represented by a single specification in the repository.
2. Size of specifications, i.e. total number of fluents representing IOPEs and total number of causal relationships that are modeled.
3. Complexity of composition heuristics, in terms of control constructs that are taken into account.

To investigate scalability in terms of these parameters, we vary the planning problem complexity in four ways: by increasing the repository size, by modifying the number of specifications that are actually considered by the planner, by increasing the complexity of service specifications and by allowing for more elaborate plans. Service specifications are synthetically generated in FLUX, each one consisting of one or two IOPE quads. We calculate separately the time required



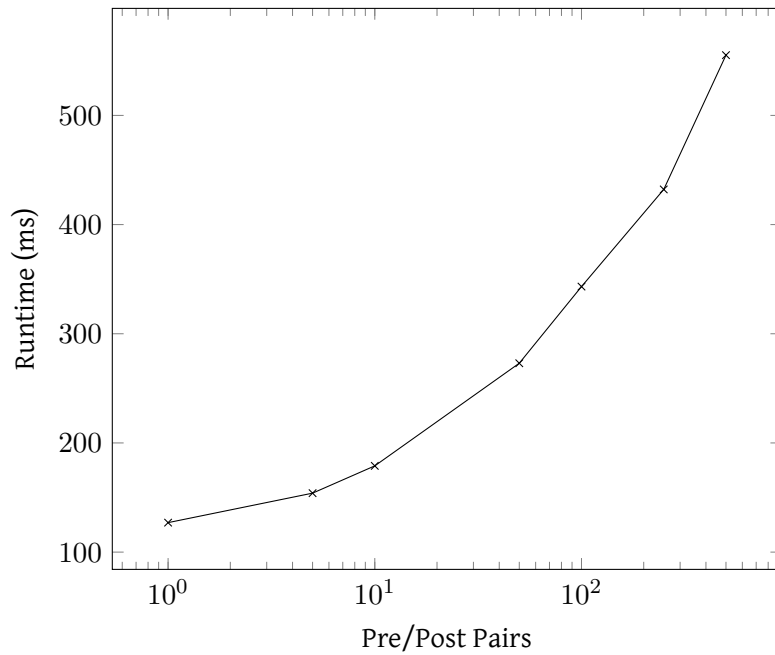


Figure 7.1: Scalability evaluation of translation process: Time

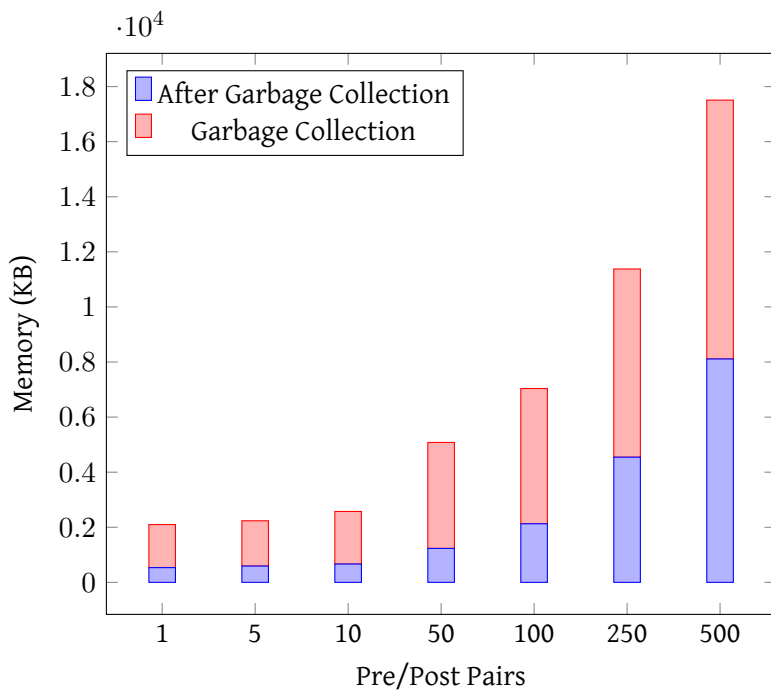


Figure 7.2: Scalability evaluation of translation process: Memory consumption

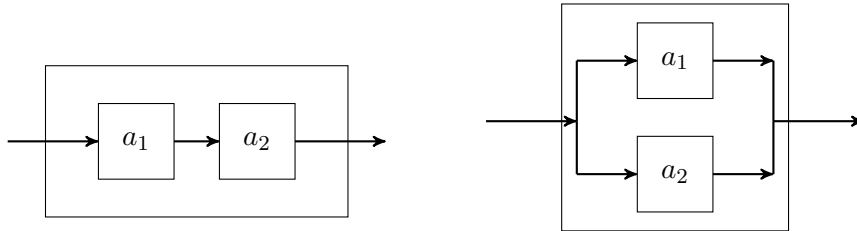
to produce a single planning solution and the time required to find all possible solutions to the planning problem.

### Specifications without Ramifications

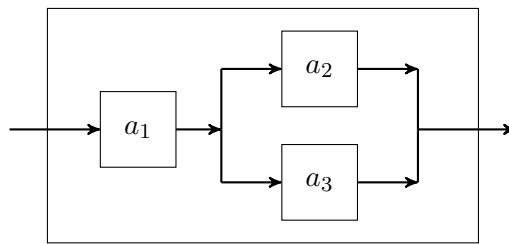
In the first experiment, we examine scalability for composition problems that involve only sequential execution, i.e. chains such as the one in Fig. 7.3a. We increase linearly the number of repository specifications that need to be considered in order to find an executable one at a given state, starting from 5 and peaking at 500. This corresponds to an increase from 10 to 2000 rules in FLUX. We examine two different cases: either all services in the repository are required for the composition goal to be achieved, or half of them, with the rest discarded early based on non-matching preconditions. Execution times are presented for both finding a single solution to the planning problem and finding all of them.

As shown in Fig. 7.4, computation time is insignificant for repositories of up to 200 specifications; for larger repositories we observe an exponential increase, with runtime remaining at reasonable levels for 500 specifications. Keeping in mind that we are dealing with repositories containing specifications and not concrete service descriptions (e.g., WSDL repositories), it is highly unlikely to encounter even more different functionalities, represented by separate specifications. Hence, the breaking point observed in this experiment is well above what is expected in real-world scenarios.

Searching for all possible planning problem solutions, instead of a single one, results in a 50% increase in computation time, on average, which is reasonable and expected, given the cost of backtracking that is required to find all solutions. As far as the effect of requiring half of the available specifications in order to achieve the composition goal, this leads to a 75% decrease, on average. This result reinforces the fact that, in real-world scenarios, the planner exhibits excellent performance: the task of finding all planning solutions for a sequential composition requiring 250 service specifications out of a repository containing 500 ones is completed in only roughly 1 second.



(a) Experiment 1 and 4: Sequential chains (b) Experiment 2 and 5: Chains of parallel executions of two services



(c) Experiment 3 and 6: Alternating between atomic services and parallel pairs

Figure 7.3: Building blocks of the experiment plans

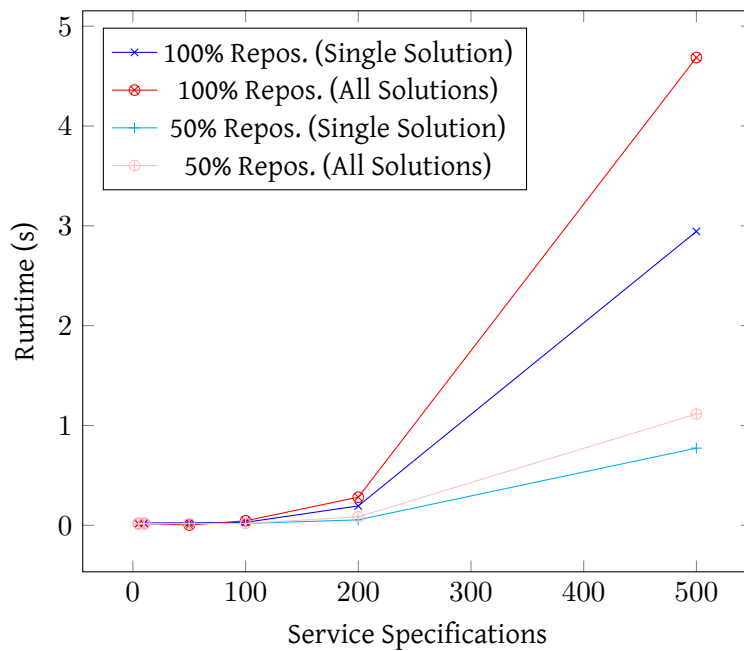


Figure 7.4: Performance results for sequential compositions

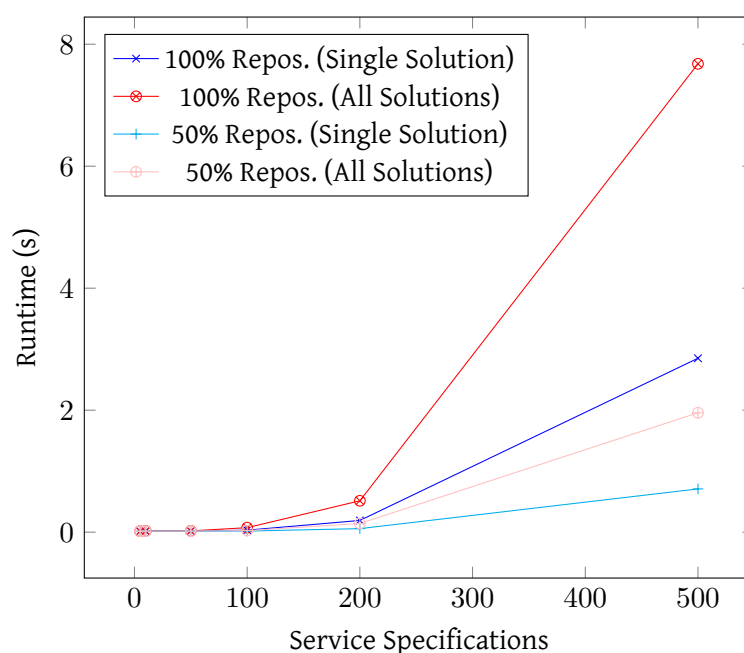


Figure 7.5: Performance results for parallel compositions

In the second experiment, we consider only parallel composition (AND-Split/Join), with consecutive copies of the composition shown in Fig. 7.3b. Again, we increase the repository size and examine the cases of requiring all or half of the contained specifications to achieve the composition goal, calculating the time required to find a single solution, as well as all possible ones. The results, shown in Fig. 7.5, indicate a behavior similar to the case of sequential composition, with one notable difference: the time for finding all solutions marks a 50% increase compared to the equivalent cases in the first experiment. This is due to the fact that, while sequence is supported without needing any special rule definition, AND Split/Join employs the special `poss_and` and `state_update_and` rules which create more backtracking points when attempting to find all solutions.

Note that, according to the analysis in Section 5.1.4, we assume that, for all AND-Split/Join pairs, no effect of a service is negated by the service executed in parallel. If this assumption does not hold, then we need to use the disjunctive form of the AND-Split/Join clause, which would lead to different experiment re-

sults, but only with regard to execution time in the case of searching for all solutions. The effect becomes more severe as we add more AND-Split/Join pairs in the plan; indicatively, for 8 consecutive pairs the time required to find all possible solutions is 0.14 seconds, while for 16 pairs, the time rises exponentially to 47.58 seconds. This is due to the fact that each new pair causes the number of possible plans to double. However, such scenarios are unrealistic, not only because the number of AND-Split/Join pairs in a given plan will rarely reach such level, but, more importantly, because it is unnatural to expect services with contradicting effects to be executed using an AND-Split/Join construct. Thus, the most effective way to approach this issue is to keep the simple non-disjunctive form of the AND-Split/Join clause and model such special cases with a more suitable construct (e.g., XOR-Split/Join or sequence).

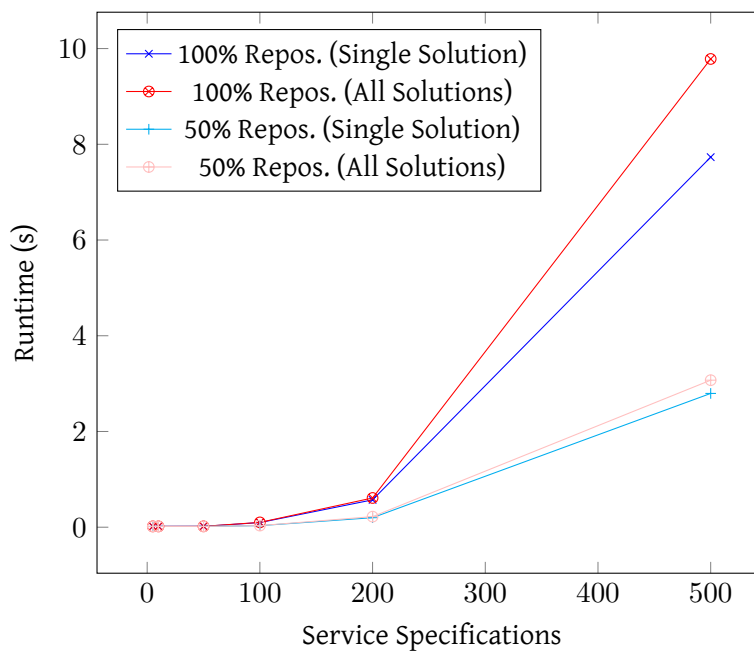


Figure 7.6: Performance results for plans with alternating sequential and parallel execution

For the third experiment, we combine the first two, creating a composition schema of alternating sequential and parallel executions, such as the one in Fig. 7.3c.

Fig. 7.6 shows that there is an almost twofold increase for the cases of 200 and 500 specifications but a much less significant increase (20% on average) when searching for all solutions. This is explained by taking into account that the composition plan is more complex than in the previous experiment, hence requiring more time to find a single solution; however the increase of backtracking points is not so severe, since we just add sequential steps to the parallel executions of the second experiment. Note that other composition schemas such as OR and XOR Split/Join, conditionals and loops are not included in our experiments due to the synthetic nature of the test sets: for any given pair of services that can be executed in a state (i.e. their *poss* clauses succeed), any of the control constructs can be considered, but only the first one defined in the heuristic will be chosen (in our case AND Split/Join), as it is the first to evaluate to true. The use of such control constructs is expected to be dictated explicitly in heuristic encodings of a planning problem, as analyzed in Section 5.1.4.

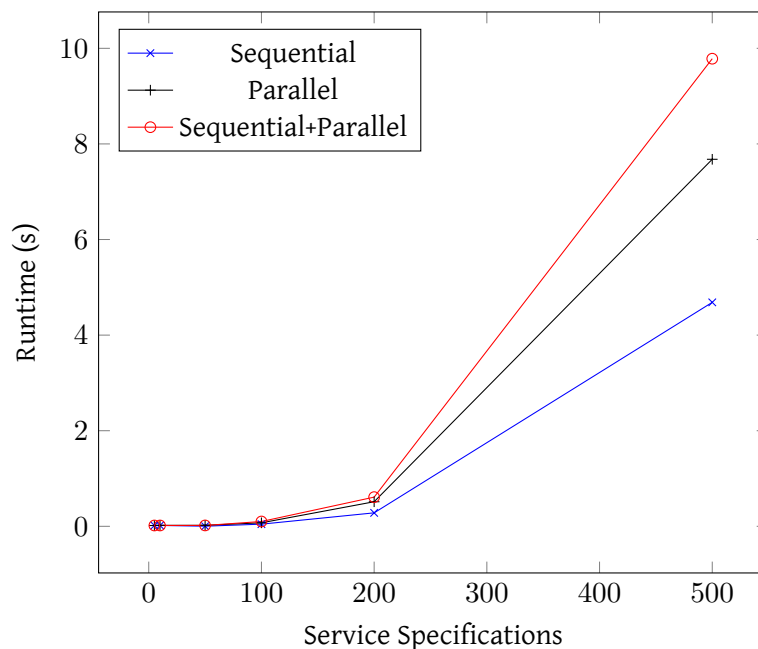


Figure 7.7: Effect of composition complexity in performance

Fig. 7.7 illustrates the effect of increasingly complex control constructs in the

planning problem, by combining, in a single graph, the most costly case of the first three experiments (finding all solutions for plans that require all service specifications in the repository). In this way, we can observe in a clearer way the effect of introducing control flow in FLUX in the worst-case scenario: there is a 50% increase in execution time for plans with sequences of parallel executions compared to sequences of atomic services and a further 20% when plans are chains of alternating sequential and parallel execution.

### **Specifications with Ramifications**

In the following three experiments, we investigate the effect of including ramifications in service specifications. We again assume a service specification repository with a size increasing from 1 to 500 (except in the case of sequential composition where it reaches 1000), and also consider the case where 50% of the specifications are required to achieve the goal. Each postcondition in half of these service specifications is associated with a ramification through the inclusion of a causal rule (a 25% increase in the size of specifications). Such an experimental setting is reasonable given the fact that, in real-world specifications, only some of them are expected to be modeled in such detail. Once again, we increase linearly the number of causal rules that are considered until a matching one is found.

As we can see in Fig. 7.8, in the worst cases there is a 30% increase in the time required to find a single solution and an 85% increase in the time required to find all possible solutions. Both increases are reasonable and are attributed to the extra effort required to find a matching causal relationship. Note that based on the discussion in Section 3.2.3, the search is stopped when a single ramification is found, since ramification chains are limited to a length of 1. Allowing larger ramification chains would definitely result in an even bigger increase in computation time.

The effect of adding ramifications to parallel composition plans is illustrated in Fig. 7.9. The increase in the single solution case is the same, around 30%. On the other hand, the increase when searching for all solutions is slightly less than

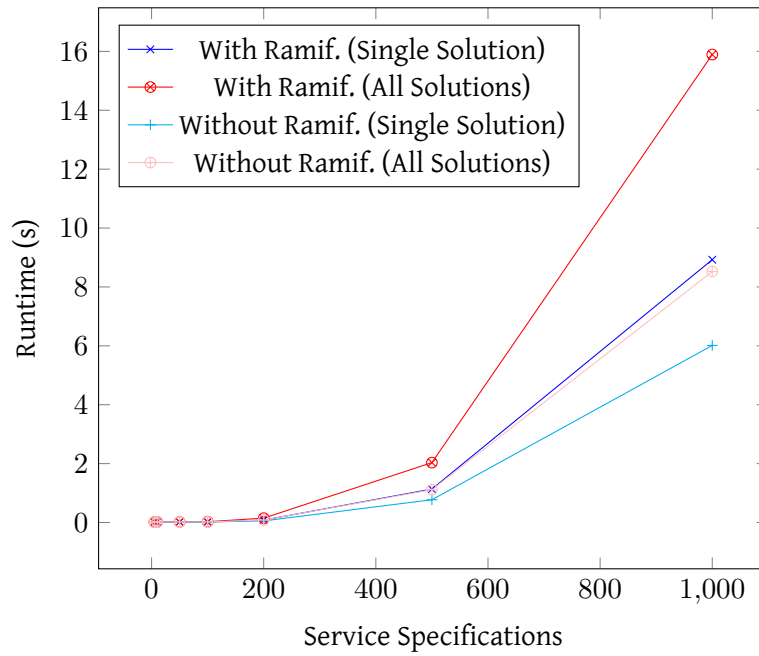


Figure 7.8: Effect of adding ramifications to sequential composition plans

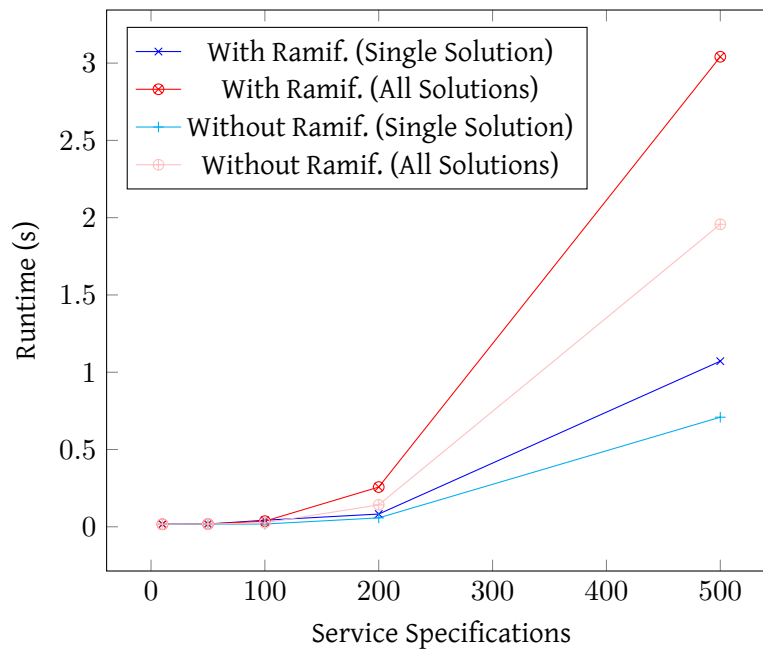


Figure 7.9: Effect of adding ramifications to parallel composition plans



the sequential case, around 60%, for the same reason as before: the special clauses that are introduced to implement AND Split/Join have already caused many backtracking points, even before introducing ramifications.

Finally, Fig. 7.10 depicts the evaluation results for the most complex case, the combination of sequential and parallel composition, including ramifications. The effect in this case is even less severe (an increase of around 15-30% in all cases), which can be attributed to the already increased complexity, rendering the addition of ramifications less significant. This is also evident in Fig. 7.11, where the results of the previous three experiments for the cost of finding all solutions are combined.

Concerning knowledge states and planning, the fact that FLUX employs negation as failure to implement knowledge (or lack thereof) of a fluent means that the cost of evaluating a `knows` or `knows_not` clause is equivalent to that of evaluating a `holds` or `not_holds` one. Hence, there is no need to conduct experiments that specifically employ knowledge clauses. The effect of arbitrary incomplete state specification using constraints is discussed in 5.2.6.

### 7.1.3 Specification-based Functional Discovery

The experimental evaluation for the specification-based discovery process is based on the following set of parameters:

1. Size of the specification repository. Again, we assume that different implementations of the same functionality are represented by a single specification in the repository.
2. Size of specifications, i.e. total number of fluents representing IOPEs. We opt not to include causal relationships, since their effect is equivalent to including more postcondition fluents, given the fact that ramifications chains longer than 1 are not allowed.
3. Matching failure position: a match fails immediately after one of the fluents

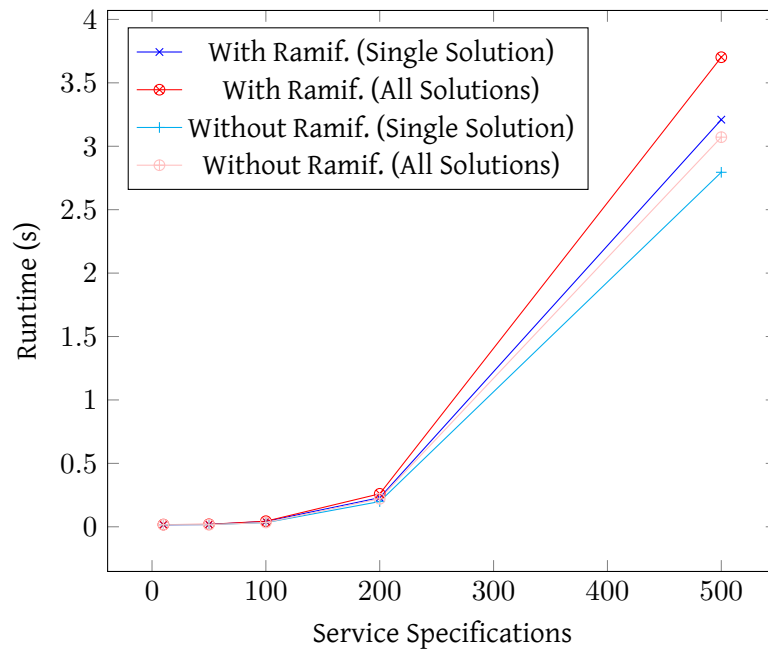


Figure 7.10: Effect of adding ramifications to plans with alternating sequential and parallel execution

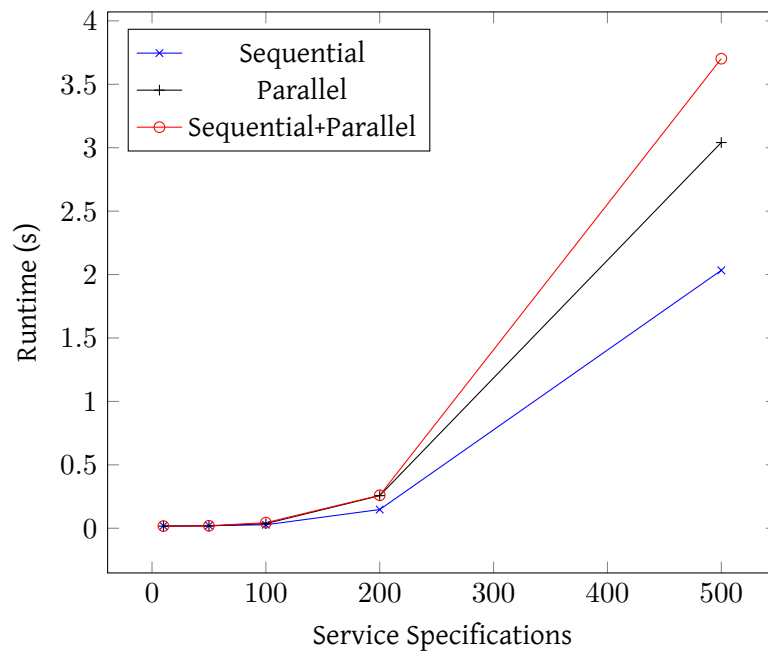


Figure 7.11: Effect of composition complexity for specifications with ramifications

that is required to hold in the source precondition is found not to hold in the target precondition. This parameter denotes the position, in the specification, of the fluent that causes failure.

In the experiment we conducted, repository size was fixed to 1000 specifications, the highest we have considered in all experiments, while we varied specification complexity by increasing the number of IOPE quads from 5 to 20. As far as failure position is concerned, we examine the two extreme cases of earliest failure (the fluent to cause failure is the first one) and latest failure (the fluent to cause failure is the last one), as well as two intermediate cases at 25% and 75% of the specification.

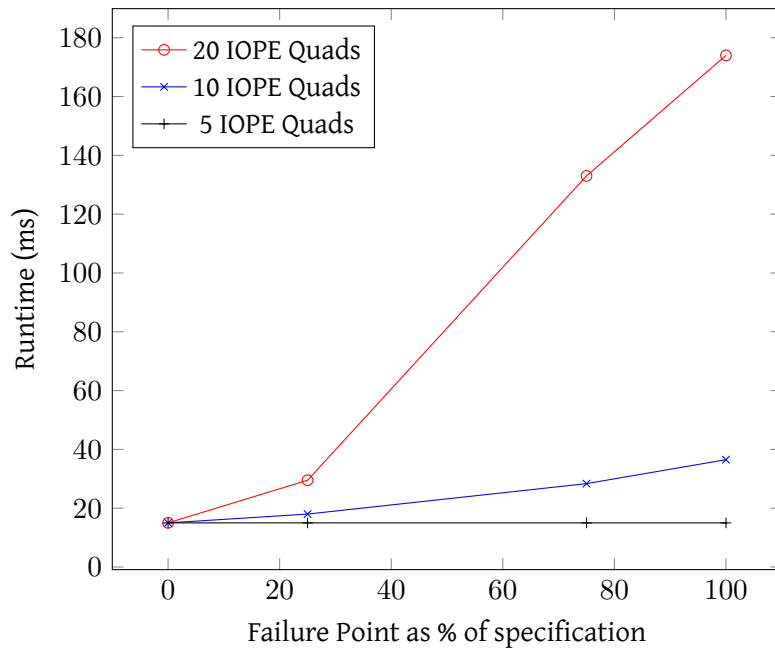


Figure 7.12: Performance evaluation for functional discovery

The results are presented in Fig. 7.12. In the case of 5 IOPE quads, which is the closest to real-world specifications, computation time is less than the minimum recordable time in *ECL<sup>i</sup>PS<sup>e</sup>*, which is 15 msec, hence we cannot obtain a more detailed view of the increase in time. Since the discovery process is expected to be executed many times, the total computation time will be a multiple of that

minimum time (see Section 7.2). For larger specifications, the effect of pushing the failure point further towards the end of the specification becomes more evident and, in the case of unnaturally large specifications of 20 IOPE quads, we observe a tenfold increase between computation time in the earliest and latest failures. This increase is justified by the fact that the complexity of the discovery process is analogous to the number of fluents (representing IOPEs) that need to be compared; the earlier the failure, the less fluents will need to be checked. Nevertheless, even in this extreme and unrealistic scenario, computation time peaks at 0.174 seconds.

#### 7.1.4 Extended Plan Pruning

In the case of plan pruning and ranking, there are multiple evaluation parameters that can be taken into consideration:

1. Number of extended plans
2. Number of tasks per plan
3. Number of implementations per task
4. Number of local goals
5. Success rate of pruning

In the first experiment, we investigate the effect of the number of tasks per plan. We considered 100 sequential plans with 20 available implementations for each task in the plan and 5 local goals to examine, while we varied the number of tasks included in each plan from 2 to 100. Note that these fixed values represent (or exceed) typical values in real-world composition design problems. We considered a success rate of 10% for pruning. As illustrated in Fig. 7.13, there is an expected moderate increase in the time required.

In the second experiment, to further increase complexity, we consider again the case of 100 different plans and 5 local goals, but also fix the size of each plan

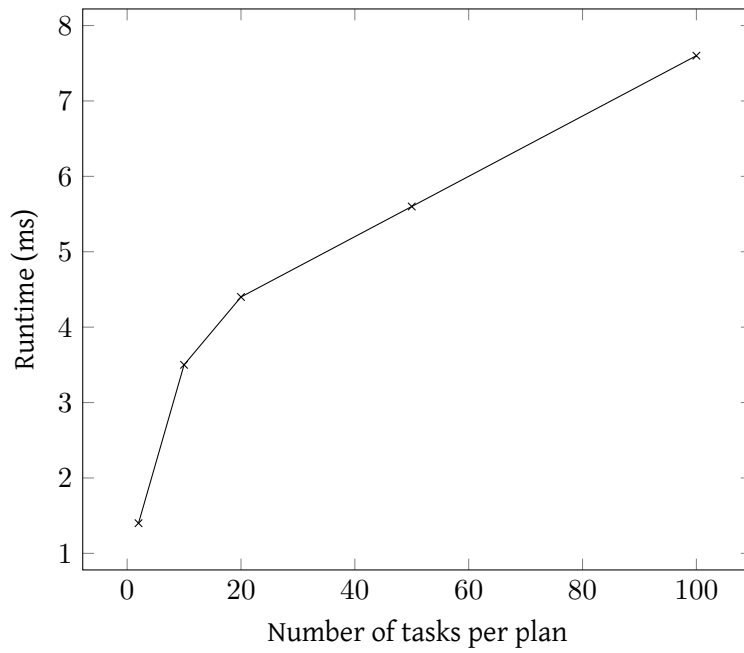


Figure 7.13: Performance of pruning for increasingly complex plans

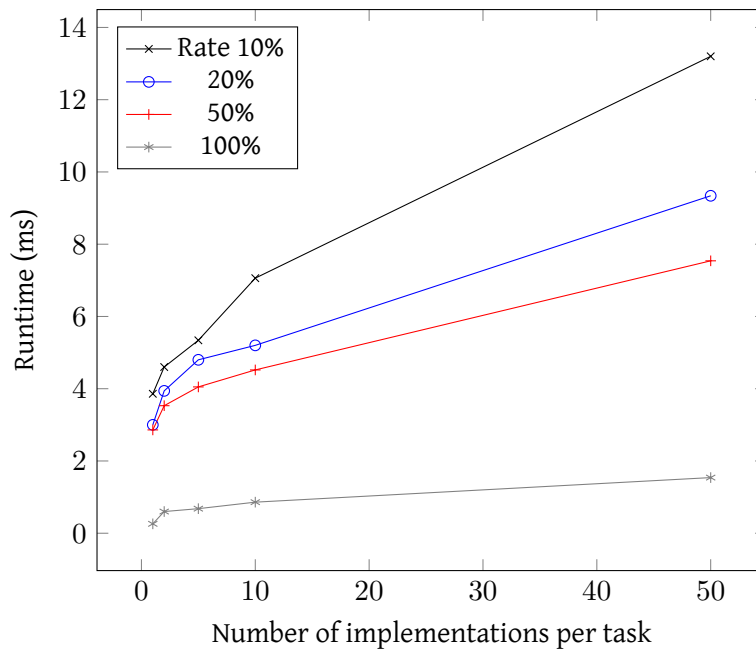


Figure 7.14: Performance of pruning for different success rates and an increasing number of implementations per task

to 100 tasks (the worst case of the first experiment), while we vary the number of available implementations for each task from 1 to 50 and the success rate for pruning from 10% to 100%, to determine the effect of these two parameters. The results, shown in Fig. 7.14, indicate again an expected increase in runtime, in a slower rate than the increase of available task implementations, while higher pruning success rates decrease runtime due to the fact that whole plans may be discarded early.

The next experiment focuses on evaluating the effect the number of extended plans has on performance. To that end, we fix the number of tasks per plan to 50, the number of implementation per task to 20, while we consider 5 local goals and a success rate of 10%. We vary the number of extended plans from 100 (the maximum value in the experiments so far) to 1000. As we observe in Fig. 7.15, increasing the number of plans from 100 to 1000 results in only a quadruple increase in computation time, which is rather efficient.

The final experiment in relation to pruning involves the number of local goals that are used in order to decide which plan to discard. Once again, we keep the rest of the parameters fixed: 500 extended plans, 50 tasks per plan, 20 implementations per task and 10% pruning success rate. The number of local goals is increased from 1 to 20. As illustrated in Fig. 7.16, the effect of considering more local goals is somewhat less significant than in the previous experiment: we observe again a quadruple increase in computation time, but in this case the number of goals has increased 20 times. This is expected since multiple goals can be checked one by one when examining the same task in a plan, while adding more plans essentially increases the number of times the whole pruning process is executed.

### 7.1.5 Extended Plan Ranking

Since the extended plan ranking process is not computationally complex, evaluation involves investigating optimality while adopting different heuristics of a problem-dependent nature, in addition to the problem-independent ones. Opti-

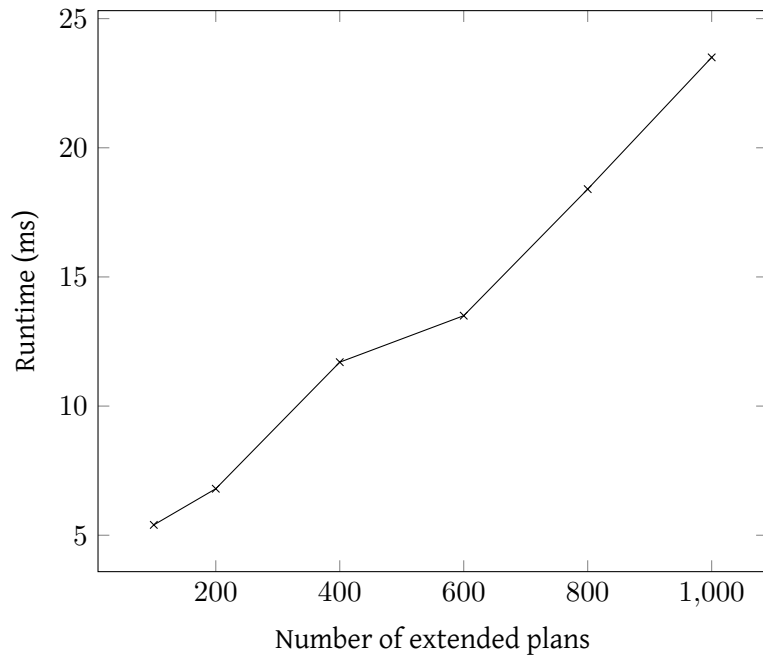


Figure 7.15: Performance of pruning for an increasing number of extended plans

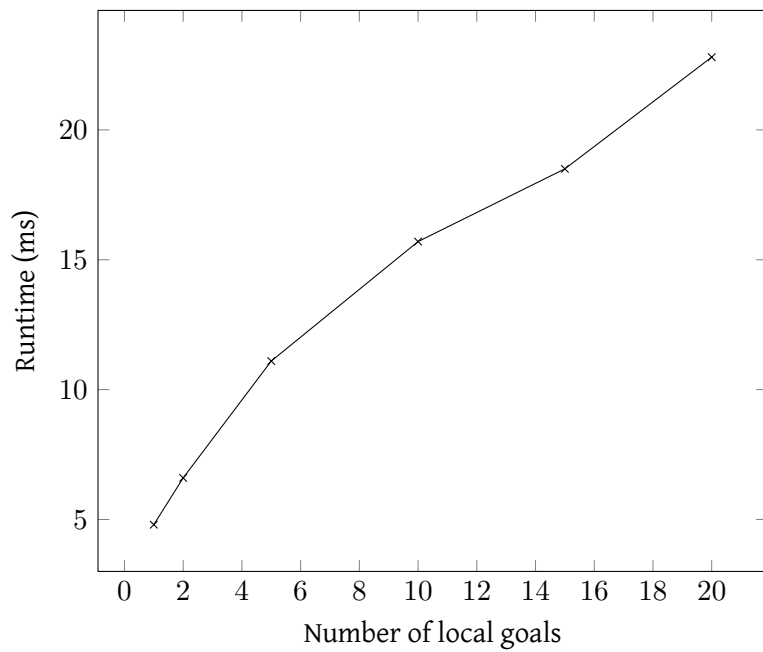


Figure 7.16: Performance of pruning for an increasing number of local goals

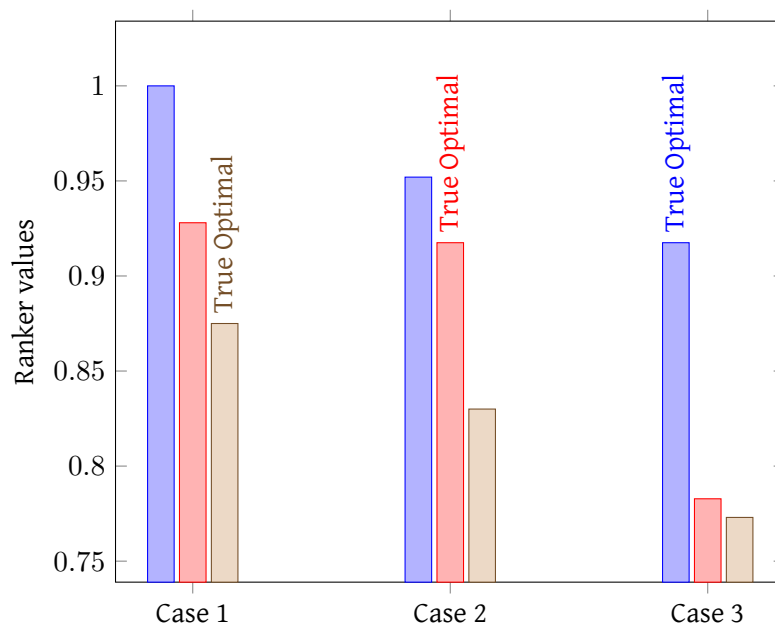


Figure 7.17: Optimality evaluation for different ranking heuristics

mality experiments require settings that refer to realistic specifications and not automatically generated ones, in order to be able to express problem-dependent criteria. To that end, we use the running example as a basis and examine the following three cases: no problem-dependent heuristics, preferring plans that support both report delivery methods, and additionally preferring that both ways of receiving a user request are provided.

As shown in Fig. 7.17, the true optimal plan is ranked third in Case 1 (no problem-dependent heuristics), deviating 12.5% from the highest value attributed by the ranker. Perfect optimality is achieved in Case 3 where both heuristics we define in relation to the composition problem are taken into account: we demand that both report delivery methods are supported, as well as both methods of receiving user requests. These results show that the ranker achieves a high-enough optimality level when using only the problem-independent heuristics concerning plan length and number of tasks; achieving perfect optimality is directly dependent on defining extra ranking criteria relying on knowledge specific to the problem at hand, as well as the expertise of the composition designer. Also note that,



given the detailed problem-dependent criteria that we have set, perfect optimality can be achieved even without taking into account problem-independent criteria. Hence, examining plan length and number of tasks makes sense only when we do not have any knowledge of problem-specific criteria.

### 7.1.6 QoS Aggregation

To evaluate the QoS aggregator, we choose to create BPMN processes synthetically, since the ones produced for the running example are not complex enough for our purposes. We evaluate the case of temporal attributes, since it is the most difficult to deal with and choose to aggregate execution time values. Two different scaling scenarios are examined. In the first, the BPMN process is a sequence with an increasing length. The results are shown in Fig. 7.18, where execution time is shown separately for the initial BPMN graph loading (using the jBPM libraries [JBoss jBPM team 2013]) and the aggregation process itself. Increase in time is analogous to the increase in sequence length, with aggregation time peaking at around 1 second and total time reaching 2.5 seconds. It should be noted that the cost of jBPM graph loading is more or less independent of the size of the plan, hence its impact in the total time is more significant in shorter plans.

The second experiment involves an AND-Split/AND-Join parallel process with an increasing number of parallel paths, each containing a single task. The results, shown in Fig. 7.19, indicate again an increase in time analogous to the increase in the number of parallel paths, with similar worst case times. Note that other control constructs need not be evaluated separately, since deterministic loops are essentially sequences of fixed length and aggregation for XOR and OR cases involves the same calculation as the AND case.

### 7.1.7 Conclusion

The individual experiments for each separate process of WSSL/CVF show, in general, a good behavior in terms of performance scalability (and optimality for

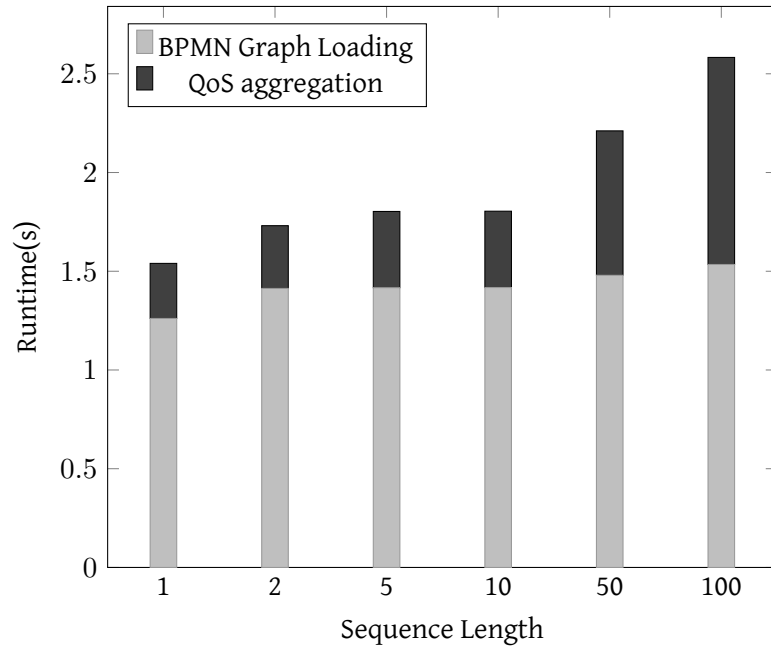


Figure 7.18: Performance evaluation for QoS aggregation (sequential case)

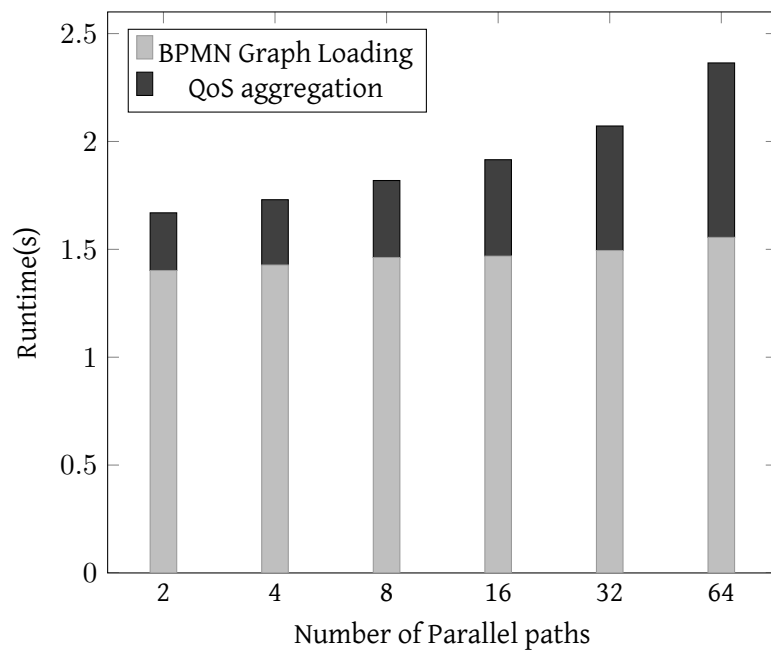


Figure 7.19: Performance evaluation for QoS aggregation (parallel case)

the case of plan ranking), even when taking into account extreme and unrealistic values for evaluation parameters, while performance is excellent when we consider parameter values that are more close to what is expected in real-world problems. More specifically:

- The WSSL planner can be considered efficient, since finding all possible plans peaks around 5 seconds for moderate repository sizes, even in the presence of ramifications (for a subset of the specifications). The breaking point can be pinpointed around repositories of 500 specifications; for larger repositories runtime exceeds 10 seconds and can be considered acceptable only in case time is not a priority (and, obviously, only for design time purposes). Such efficiency and termination are only guaranteed if heuristics are defined for planning; in the general case where no heuristics are defined, planning cannot be considered efficient.
- Specification-based functional discovery is inexpensive for a single task, hence it can be executed for all tasks of large plans, containing tens to hundreds of tasks, without observing a significant rise in total computation time.
- The pruning process is the least expensive of all phases in the proposed framework, since it peaks at around 20msec in the worst possible case.
- Optimality in the ranking process relies heavily on heuristics that are specific to each problem, hence its success depends mostly on the expertise of the composition designer.
- More than half of the runtime for the aggregation process is spent on the graph loading process by the jBPM libraries and the overall cost is around 2 seconds. In the following section, we also consider the alternative aggregation mechanism, proposed in [Mello Ferreira et al. 2009].

## 7.2 Overall Evaluation

In this section, we design and conduct a performance evaluation for the complete WSSL/CVF framework, again based on synthetic specifications. We designed a service specification repository of a reasonable size and complexity, containing 200 distinct specifications, of which: 45% contain a single IOPE quad, 30% contain 2 IOPE quads, 15% contain 3 IOPE quads, 7% contain 4 IOPE quads and 3% contain 5 IOPE quads. 50% of all specifications in the repository contain a causal relationship linking one postcondition with one ramification. The specifications are produced in such a way that they lead to sequential chains (a service's output and postcondition are the next service's input and precondition).

We select 5 increasingly complex composition goals that form the 5 different cases that we examine in this evaluation. More specifically:

**Case 1** 3 plans of length 9 are returned by the planner in less than 0.0155sec

**Case 2** 5 plans of length 9-10 are returned by the planner in less than 0.0156sec

**Case 3** 20 plans of length 9-12 are returned by the planner in 0.0158sec

**Case 4** 112 plans of length 10-14 are returned by the planner in 0.0483sec

**Case 5** 2167 plans of length 10-17 are returned by the planner in 0.85105sec

Following the planning process, we evaluate the functional discovery process for each of the 5 different cases. For the sake of simplification, we assume that each one of the specifications contains 2 IOPE quads instead of ranging from 1 to 5. This is an average representation of the initial repository, since 45% of the rest of the services are smaller and thus discovery for them is faster but 25% are larger and lead to slower discovery times. We also assume the average case where the failure point is halfway through the specification and consider a success rate of 90% for all functional discovery executions; we also distribute discovery failures in such a way that they result in discarding 10% of all abstract plans in each case, due to the fact that a single task was not matched with any concrete implementation.

Table 7.1: Performance of functional discovery for different cases of overall evaluation

Cases	Discovery Cost	# Extended Plans
Case 1	0.107694	3
Case 2	0.195316	5
Case 3	0.215388	18
Case 4	0.235331	101
Case 5	0.255274	1951

In order to obtain multiple concrete implementations for each task, we assume that for each distinct specification in the repository, there are 1-4 other identical ones. Given the fact that each composition is a chain of services that is a subset of the repository and examining the different specifications of the plans that form each category, we result in the following maximum number of runs for the functional discovery process: 27 for Case 1, 49 for Case 2, 54 for Case 3, 59 for Case 4 and 64 for Case 5. Table 7.1 contains the cost of the functional discovery process for each of the five different cases as well as the number of the extended plans produced (the values are rounded up). Note that we execute discovery for groups of 3 specifications, so that we surpass the minimum recordable time in *ECL<sup>i</sup>PS<sup>e</sup>*.

We assume that after discovery, each task is associated with 1 to 5 concrete implementations with the following probabilities: 45% yields 1 implementation, 30% yields 2 implementations, 15% yields 3 implementations, 7% yields 4 implementations and 3% yields 5 implementations. Assuming 5 local goals, this results in the following sets of extended plans, associated with the computation times for pruning and ranking:

**Case 1** 3 plans with 9 tasks, of which 4 are linked with 1 implementation, 3 with 2, 1 with 3 and 1 with 4. Pruning and ranking costs 0.003sec.

- Case 2** 5 plans with 10 tasks, of which 4 are linked with 1 implementation, 3 with 2, 1 with 3, 1 with 4 and 1 with 5. Pruning and ranking costs 0.0033sec.
- Case 3** 18 plans with 11 tasks, of which 5 are linked with 1 implementation, 3 with 2, 1 with 3, 1 with 4 and 1 with 5. Pruning and ranking costs 0.0054sec.
- Case 4** 101 plans with 12 tasks, of which 5 are linked with 1 implementation, 4 with 2, 1 with 3, 1 with 4 and 1 with 5. Pruning and ranking costs 0.009sec.
- Case 5** 1951 plans with 14 tasks, of which 6 are linked with 1 implementation, 4 with 2, 2 with 3, 1 with 4 and 1 with 5. Pruning and ranking costs 0.0354sec.

Since the plans are synthetically created, the ranking process can only take into account problem-independent criteria. Also, since all plans are sequences, the only criterion that is relevant is the plan length. Hence, the ranking process is trivial and there is no need to calculate any additional cost. The total computation time values corresponding to the three first phases of the overall process, for the 5 different cases that we examine, are shown in Fig. 7.20.

The resulting optimal extended plans that are fed to the final phase of the framework, the QoS-based selection process, are the following:

- Cases 1-3** Sequential plan with 9 tasks, with 3 concrete implementations (on average) for each task, assuming 3 different QoS profiles for each implementation.
- Cases 4-5** Sequential plan with 10 tasks, with 3 concrete implementations (on average) for each task, assuming 3 different QoS profiles for each implementation.

For QoS-based selection, we employ the algorithms defined in [Mello Ferreira et al. 2009]. We vary the QoS global goals from 1 to 20 and assume that each concrete service offers 3 different QoS profiles (representing low, medium and high quality levels). The overall results for the complete composition process (all four phases) of the proposed framework for all cases are shown in Fig. 7.21.

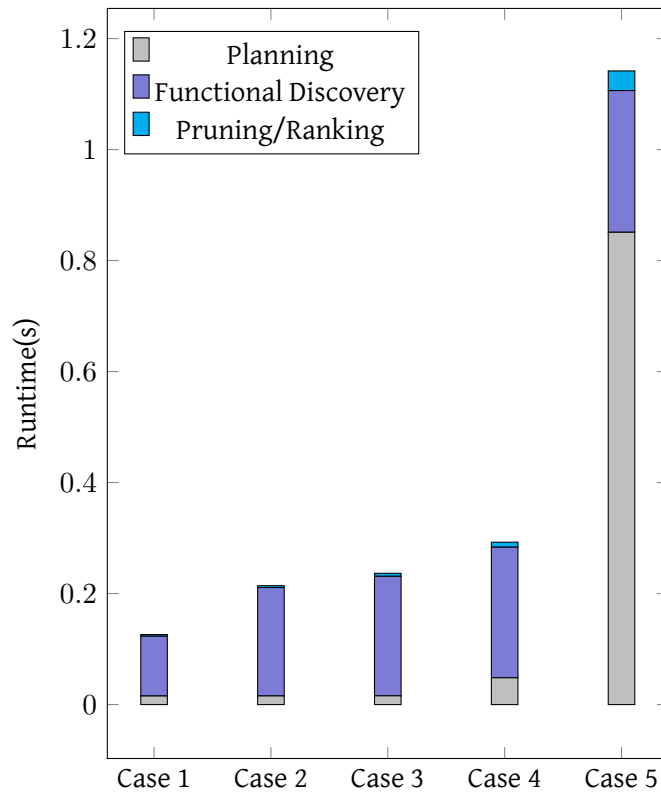


Figure 7.20: Performance results for the first three phases of WSSL/CVF

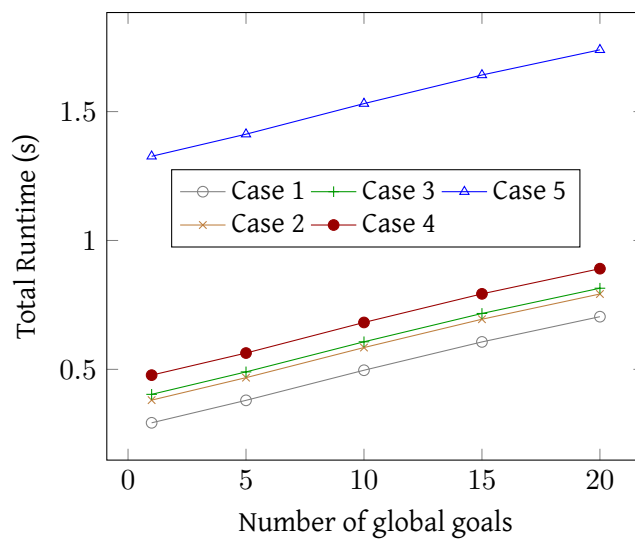


Figure 7.21: Overall performance for increasing numbers of global QoS goals

As we can deduce from Figs. 7.20 and 7.21, the vast majority of the computation time, with the exception of Case 5, is attributed to the functional discovery process, while the pruning/ranking cost is almost non-existent in comparison. This is due to the fact that discovery has to be run for each distinct abstract task contained in the plans and we assume the worst case (the maximum number of distinct tasks per plan). In Case 5, the complexity of the problem results in a 20-time increase of the planning computation time. Nevertheless, the overall computation cost, at the worst case, peaks at 1.74 seconds, which is rather satisfactory, strengthening the argument that WSSL/CVF can be of great assistance to composition designers by offering a semi-automatic way of navigating through the numerous choices and decisions that a composition problem includes.

As analyzed in Section 5.2.5, there are several prerequisites, related to the expertise of SBA designers and composition architects, that need to be satisfied in order for WSSL/CVF to produce the results that were shown in this Chapter. To strengthen the validity of these results, a second set of evaluation experiments needs to be carried out, focusing on usability and with the participation of actual users with varying levels of expertise in service design. Due to the difficulty in finding such candidates in either academic or industry partners, usability evaluation is left as a topic for future work.



## Chapter 8

# Conclusions and Future Research

### Contents

---

8.1 Synopsis of Contributions . . . . .	195
8.2 Directions for Future Research . . . . .	197

---

### 8.1 Synopsis of Contributions

The primary objective of the research behind this thesis was to motivate the need for service specifications that take into account the frame, ramification and qualification problems, by illustrating the effects these problems have in service science, especially when describing and composing services. This led to the definition of the proposed language, WSSL, which advances service research by being the only specification and composition language that not only addresses the aforementioned representation problems but manages to support many desired features in service description and composition in a unified and integrated manner.

WSSL allows service designers to accurately express the behavior of a service, by defining conditions before and after execution, expressing or inferring causal relationships among conditions and accounting for unexpected unsuccessful executions, resulting in an all-encompassing specification of the precise way a ser-

vice behaves under any possible circumstance. Any information contained in a WSSL specification can be linked to an ontology concept, resulting in semantics-aware content. Further extensions of the language instill the ability to define QoS profiles, to model service compositions using any fundamental composition pattern and to handle partially observable states. WSSL essentially makes a clear distinction between service description in the traditional form that is offered by current languages such as WSDL, OWL-S and WSMO and service specification, in the sense of a complete model of service behavior.

Such rich behavior specifications can then be exploited to facilitate a multitude of actions throughout the life-cycle of a service or an SBA, whether it is verifying a service against a WSSL specification, discovering a service that conforms to a WSSL specification, composing services in order to achieve a set of requirements expressed in WSSL and verifying the resulting composition, or even using WSSL to express the goals of service adaptation. The proposed framework acts as proof of that claim, by presenting a service composition and verification approach that simultaneously achieves a unique combination of desired requirements.

WSSL/CVF can be used as a powerful tool by service designers, providing them with an automated way of creating composite processes that exhibit a behavior defined by a series of functional and non-functional (e.g., QoS-based) properties, solving any composition problem independent of a specific domain. Since this behavior is expressed using WSSL, it may take into account ramifications and accidental qualifications while information may also be semantically annotated. The resulting composite processes can either be abstract (i.e. specification-based), in order to be dynamically bound to actual service endpoints at runtime, or can be concretized based on the defined QoS properties. Moreover, the generated processes can contain multiple composition patterns, including non-deterministic ones, namely conditional and iterative execution. Also, due to the fact that WSSL handles partial observability, the composition process can return results even under partial knowledge of the initial state.

The applicability of the proposed framework in realistic settings is backed up

by the experimental evaluation that we conducted. The experiments prove that all phases of the composition process scale well and can solve composition problems of high complexity in an efficient manner. This lends credibility to the fact that increased expressiveness does not always compromise efficiency, thus enabling WSSL and the accompanying framework to be useful in assisting service designers to solve practical, real-world composition problems.

## 8.2 Directions for Future Research

This dissertation paves the way for several possible directions for future research, some of which are outlined in this section. With regard to usability, it would be helpful to create a WSSL tool set that provides an interface for users to create WSSL specifications either from scratch or by translating existing descriptions in WSDL, OWL-S or WSMO and filling up information that is exclusive to WSSL. Mechanisms to derive missing information, or information that the provider may not be willing to provide can also be offered. Also, the user should be able to select varying levels of information completeness, from simplified WSSL specifications that offer only basic behavior description to specifications that support the full capabilities set of WSSL and its extensions.

Additionally, the existing framework can be extended with a visualization component that allows service designers to view resulting compositions (e.g., in the form of BPMN processes), modify them according to their expert knowledge and execute them (e.g., on a BPMN engine). Finally, a user study can be conducted in order to evaluate the modeling effort and usability of WSSL and the accompanying tool set, as well as effectiveness and usability of WSSL/CVF. This study can indicatively focus on the ability of users with varying levels of expertise in creating atomic WSSL specifications, with and without the envisioned tool set, as well as the level of effort required in order to define suitable heuristic encodings and ranking criteria for varying composition problems.

Concerning the existing functionality of WSSL/CVF, it is interesting to ex-

explore alignment and normalization procedures for functional and non-functional aspects. For instance, WSSL specifications may refer to the same concept but using different ontologies or WSSL quality profiles may refer to different QoS models, where different names or different metrics are used for the same QoS attribute. Additionally, ways to optimize the functional discovery process can be investigated, e.g., by keeping a result cache, so that discovery is not run multiple times for the same service, while also exploring cases other than exact matching, such as subsumption or plug-in matches. Also, the framework can be adapted to support soft conditions and constraints [Meseguer et al. 2006] (either functional or non-functional), i.e. constraints whose violation does not lead to the associated service (and, in turn, the overall plan) being discarded. Moreover, QoS decomposition approaches, such as the one in [Alrifai et al. 2012], can be considered in order to reduce overall composition time. Finally, we can explore whether combining functional and non-functional composition in a single step is effective and under what circumstances.

A further research direction is to introduce adaptation features to the proposed composition framework. Violations that trigger adaptation may occur due to various events, from infrastructure failures to errors related to functionality and can be detected using service monitoring techniques. These different sources of violation bring about the demand for a cross-layer treatment of both monitoring and adaptation. One suitable line of work that can be considered in order to realize proactive cross-layer monitoring and adaptation is that of [Zeginis et al. 2012] and [Zeginis et al. 2013]. In cases where adaptation is not effective, the framework can be extended to support re-planning in order to partially recreate the plan based on a modified goal specification.

Finally, another significant research direction is to investigate whether the proposed service specification and composition approaches can be applied in Cloud environments and what are the prerequisites for this to be realized. This would bring the ideas of behavior specification and formality in service description to the Cloud Computing setting. For instance, WSSL can be extended in order to sup-

port cloud service specifications that include deployment information. In turn, the composition framework can be integrated into a Cloud deployment framework where, based on a set of deployed requirements, the resulting composite processes are deployed on Cloud providers, in Single-Cloud or Multi-Cloud settings, following the initial exploration performed in [Baryannis et al. 2013].



# Bibliography

- AGARWAL, V., CHAFLE, G., MITTAL, S., AND SRIVASTAVA, B. 2008. Understanding Approaches for Web Service Composition and Execution. In *Bangalore Compute Conf.*, R. K. Shyamasundar, Ed. ACM, 1.
- AKKIRAJU, R., FARRELL, J., J.MILLER, NAGARAJAN, M., SCHMIDT, M., SHETH, A., AND VERMA, K. 2005. Web Service Semantics - WSDL-S, A joint UGA-IBM Technical Note, version 1.0. Tech. rep., UGA-IBM, April 2005.
- AKKIRAJU, R., VERMA, K., GOODWIN, R., DOSHI, P., AND LEE, J. 2004. Executing Abstract Web Process Flows. In *Proceedings of the ICAPS Workshop on Planning and Scheduling for Web and Grid Services*. AAAI Press, 9–15.
- ALI, S. A., ROOP, P. S., WARREN, I., AND BHATTI, Z. E. 2011. Unified Management of Control Flow and Data Mismatches in Web Service Composition. In *SOSE*, J. Z. Gao, X. Lu, M. Younas, and H. Zhu, Eds. IEEE, 93–101.
- ALRIFAI, M., RISSE, T., AND NEJDL, W. 2012. A Hybrid Approach for Efficient Web Service Composition with End-to-end QoS Constraints. *ACM Trans. Web* 6, 2, 7:1–7:31.
- APT, K. R. AND WALLACE, M. 2007. *Constraint Logic Programming Using ECLiPSe*. Cambridge University Press.
- ARDAGNA, D., COMUZZI, M., MUSSI, E., PERNICI, B., AND PLEBANI, P. 2007. PAWS: A Framework for Executing Adaptive Web-Service Processes. *IEEE Software* 24, 39–46.

- BALZER, S., LIEBIG, T., AND WAGNER, M. 2004. Pitfalls of OWL-S: A Practical Semantic Web Use Case. In *Proceedings of the 2nd International Conference on Service Oriented Computing*. ICSOC '04. ACM, 289–298.
- BARAKAT, L., MILES, S., POERNOMO, I., AND LUCK, M. 2011. Efficient Multi-granularity Service Composition. In *ICWS*. IEEE Computer Society, 227–234.
- BARESI, L., BIANCULLI, D., GHEZZI, C., GUINEA, S., AND SPOLETINI, P. 2007. Validation of Web Service Compositions. *IET Software* 1, 6, 219–232.
- BARTALOS, P. AND BIELIKOVÁ, M. 2012. Effective QoS Aware Service Composition Based on Forward Chaining with Service Space Restriction. In *Semantic Web Services*, B. Blake, L. Cabral, B. König-Ries, U. Küster, and D. Martin, Eds. Springer Berlin Heidelberg, 313–328.
- BARYANNIS, G., CARRO, M., DANYLEVYCH, O., DUSTDAR, S., KARASTOYANOVA, D., KRITIKOS, K., LEITNER, P., ROSENBERG, F., AND WETZSTEIN, B. 2008. PO-JRA-2.2.1: Overview of the State of the Art in Composition and Coordination of Services. Tech. rep., S-Cube Network of Excellence, July 2008.
- BARYANNIS, G., CARRO, M., AND PLEXOUSAKIS, D. 2012. Deriving Specifications for Composite Web Services. In *COMPSAC*. IEEE Computer Society, 432–437.
- BARYANNIS, G., DANYLEVYCH, O., KARASTOYANOVA, D., KRITIKOS, K., LEITNER, P., ROSENBERG, F., AND WETZSTEIN, B. 2010. Service Composition. In *Service Research Challenges and Solutions for the Future Internet*, M. P. Papazoglou, K. Pohl, M. Parkin, and A. Metzger, Eds. Lecture Notes in Computer Science Series, vol. 6500. Springer Berlin Heidelberg, 55–84.
- BARYANNIS, G., GAREFALAKIS, P., KRITIKOS, K., MAGOUTIS, K., PAPAIOANNOU, A., PLEXOUSAKIS, D., AND ZEGINIS, C. 2013. Lifecycle Management of Service-based Applications on Multi-Clouds: A Research Roadmap. In *Proceedings of the International Workshop on Multi-Cloud Applications and Federated Clouds (MultiCloud 2013)*. ACM, 13–20.



- BARYANNIS, G., KRITIKOS, K., AND PLEXOUSAKIS, D. 2014. A Specification-based, QoS-Aware Service Composition and Verification Framework. Submitted to IEEE Transactions on Services Engineering.
- BARYANNIS, G. AND PLEXOUSAKIS, D. 2010a. Automated Web Service Composition: State of the Art and Research Challenges. Tech. Rep. ICS-FORTH/TR-409, Foundation for Research and Technology - Hellas, October 2010.
- BARYANNIS, G. AND PLEXOUSAKIS, D. 2010b. Towards Realizing Dynamic QoS-aware Web Service Composition. In *Proceedings of the PhD Symposium at the 8th IEEE European Conference on Web Services*, W. Zimmermann, Ed. Institute of Computer Science, University Halle-Wittenberg, 37–40.
- BARYANNIS, G. AND PLEXOUSAKIS, D. 2013. WSSL: A Fluent Calculus-Based Language for Web Service Specifications. In *CAiSE 2013*, C. Salinesi, M. C. Norrie, and Ó. Pastor, Eds. LNCS Series, vol. 7908. Springer Berlin Heidelberg, 256–271.
- BARYANNIS, G. AND PLEXOUSAKIS, D. 2014. Fluent Calculus-based Semantic Web Service Composition and Verification using WSSL. In *ICSOC 2013 Workshops*, A. Lomuscio et al., Eds. LNCS Series, vol. 8377. Springer International Publishing Switzerland, 256–270.
- BATTLE, S., BERNSTEIN, A., BOLEY, H., GROSOFF, B., GRUNINGER, M., HULL, R., KIFER, M., MARTIN, D., MCILRAITH, S., MCGUINNESS, D., SU, J., AND TABET, S. 2005. Semantic Web Services Framework (SWSF) Overview. World Wide Web Consortium, Member Submission SUBM-SWSF-20050909.
- BEAUCHE, S. AND POIZAT, P. 2008. Automated Service Composition with Adaptive Planning. In *Service-Oriented Computing - ICSOC 2008*, A. Bouguettaya, I. Krüger, and T. Margaria, Eds. Lecture Notes in Computer Science Series, vol. 5364. Springer Berlin Heidelberg, 530–537.
- BENBERNOU, S., CAVALLARO, L., HACID, M. S., KAZHAMIKIN, R., KECSKEMETI, G., PAZAT, J.-L., SILVESTRI, F., UHLIG, M., AND WETZSTEIN, B. 2008. PO-JRA-1.2.1:

- State of the Art Report, Gap Analysis of Knowledge on Principles, Techniques and Methodologies for Monitoring and Adaptation of SBAs. Tech. rep., S-Cube Network of Excellence, July 2008.
- BENNETT, K., LAYZELL, P., BUDGEN, D., BRERETON, P., MACAULAY, L., AND MUNRO, M. 2000. Service-Based Software: The Future for Flexible Software. In *Seventh Asia-Pacific Software Engineering Conference (APSEC2000)*. IEEE, 214–221.
- BERARDI, D., CALVANESE, D., GIACOMO, G. D., HULL, R., AND MECELLA, M. 2005a. Automatic Composition of Transition-based Semantic Web Services with Messaging. In *VLDB*, K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P.-A. Larson, and B. C. Ooi, Eds. ACM, 613–624.
- BERARDI, D., CALVANESE, D., GIACOMO, G. D., LENZERINI, M., AND MECELLA, M. 2005b. Automatic Service Composition Based on Behavioral Descriptions. *Int. J. Cooperative Inf. Syst.* 14, 4, 333–376.
- BERTOLI, P., KAZHAMIKIN, R., PAOLUCCI, M., PISTORE, M., RAIK, H., AND WAGNER, M. 2009. Control Flow Requirements for Automated Service Composition. In *ICWS '09: Proceedings of the 2009 IEEE International Conference on Web Services*. IEEE Computer Society, 17–24.
- BERTOLI, P., PISTORE, M., AND TRAVERSO, P. 2010. Automated Composition of Web Services by Planning in Asynchronous Domains. *Artif. Intell.* 174, 3-4, 316–361.
- BESNARD, P., QUINIOU, R., AND QUINTON, P. 1983. A Theorem-Prover for a Decidable Subset of Default Logic. In *AAAI*, M. R. Genesereth, Ed. AAAI Press, 27–30.
- BHUVANESWARI, A. AND KARPAGAM, G. R. 2010. Applying Fluent Calculus for Automated and Dynamic Semantic Web Service Composition. In *Proceedings of the 1st International Conference on Intelligent Semantic Web-Services and Applications (ISWSA 2010)*. ACM.
- BIANCULLI, D., GHEZZI, C., SPOLETINI, P., BARESI, L., AND GUINEA, S. 2008. A Guided Tour through SAVVY-WS: A Methodology for Specifying and Validating Web

- Service Compositions. In *Advances in Software Engineering*, E. Börger and A. Cisternino, Eds. Lecture Notes in Computer Science Series, vol. 5316. Springer Berlin Heidelberg, 131–160.
- BIRON, P. V. AND MALHOTRA, A. 2004. XML Schema Part 2: Datatypes Second Edition. Tech. rep., W3C Recommendation, October 2004.
- BOLEY, H., ATHAN, T., PASCHKE, A., TABET, S., GROSOFF, B., BASSILIADES, N., GOVERNATORI, G., OLKEN, F., AND HIRTLE, D. 2012. Schema Specification of Deliberation RuleML Version 1.0. <http://ruleml.org/1.0/>.
- BOOTH, D., HAAS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M., FERRIS, C., AND ORCHARD, D. 2004. Web Services Architecture. W3C Note NOTE-ws-arch-20040211, World Wide Web Consortium, February 2004.
- BORGIDA, A., MYLOPOULOS, J., AND REITER, R. 1995. On the Frame Problem in Procedure Specifications. *Software Engineering, IEEE Transactions on* 21, 10, 785–798.
- BOX, D., EHNEBUSKE, D., KAKIVAYA, G., LAYMAN, A., MENDELSON, N., NIELSEN, H. F., THATTE, S., AND WINER, D. 2000. Simple Object Access Protocol (SOAP) 1.1. W3C Note, World Wide Web Consortium, May 2000.
- BROGI, A. AND CORFINI, S. 2008. Ontology- and Behavior-Aware Discovery of Web Service Compositions. *Int. J. Cooperative Inf. Syst.* 17, 3, 319–347.
- CARMAN, M., SERAFINI, L., AND TRAVERSO, P. 2003. Web Service Composition as Planning. In *Proceedings of the ICAPS 2003 Workshop on Planning for Web Services*. AAAI Press.
- CHAN, K. S. M., BISHOP, J., AND BARESI, L. 2006. Survey and Comparison of Planning Techniques for Web Services Composition. Tech. rep., Department of Computer Science, University of Pretoria, 0002 Pretoria, South Africa, December 2006.
- CHIFU, V. R., SALOMIE, I., HARSA, I., AND GHERGA, M. 2009. Semantic Web Service Composition Method Based on Fluent Calculus. In *Proceedings of the 11th*

- International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2009)*, S. M. Watt, V. Negru, T. Ida, T. Jebelean, and D. Petcu, Eds. IEEE Computer Society, 325–332.
- CHINNICI, R., MOREAU, J.-J., RYMAN, A., AND WEERAWARANA, S. 2007. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. World Wide Web Consortium, Recommendation REC-wsdl20-20070626.
- DOHERTY, P. 1994. Reasoning about Action and Change Using Occlusion. In *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI 1994)*. Wiley, 401–405.
- DRANIDIS, D., RAMOLLARI, E., AND KOURTESIS, D. 2009. Run-time Verification of Behavioural Conformance for Conversational Web Services. In *Web Services, 2009. ECOWS '09. Seventh IEEE European Conference on*. IEEE, 139–147.
- DUERST, M. AND SUIGNARD, M. 2005. Internationalized Resource Identifiers (IRIs). RFC 3987.
- DUSTDAR, S. AND SCHREINER, W. 2005. A survey on web services composition. *IJWGS* 1, 1, 1–30.
- FARRELL, J. AND LAUSEN, H. 2007. Semantic Annotations for WSDL and XML Schema. World Wide Web Consortium, Recommendation REC-sawSDL-20070828.
- FIELDING, R. T. AND TAYLOR, R. N. 2002. Principled Design of the Modern Web Architecture. *ACM Transactions on Internet Technologies* 2, 2, 115–150.
- FISTEUS, J., FERNÁNDEZ, L., AND KLOOS, C. 2004. Formal Verification of BPEL4WS Business Collaborations. In *Proceedings of the 5th International Conference on Electronic Commerce and Web Technologies (EC-Web '04)*, K. Bauknecht, M. Bichler, and B. Proll, Eds. Lecture Notes in Computer Science Series, vol. 3182. Springer-Verlag, Berlin, 79–94.

- FOSTER, H., UCHITEL, S., MAGEE, J., AND KRAMER, J. 2003. Model-based Verification of Web Service Composition. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 152–161.
- FOSTER, H., UCHITEL, S., MAGEE, J., AND KRAMER, J. 2004. Compatibility Verification for Web Service Choreography. In *ICWS (2004-09-23)*. IEEE Computer Society, 738–741.
- FUJII, K. AND SUDA, T. 2006. Semantics-Based Dynamic Web Service Composition. *Int. J. Cooperative Inf. Syst.* 15, 3, 293–324.
- FUJII, K. AND SUDA, T. 2009. Semantics-based Context-aware Dynamic Service Composition. *ACM Trans. Auton. Adapt. Syst.* 4, 2, 12:1–12:31.
- GALTON, A. 1990. *Logic for Information Technology*. Wiley.
- GHALLAB, M., ISI, C. K., PENBERTHY, S., SMITH, D. E., SUN, Y., AND WELD, D. 1998. PDDL - The Planning Domain Definition Language. Tech. rep., CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, October 1998.
- GHALLAB, M., NAU, D., AND TRAVERSO, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann Publishers.
- GIACOMO, G. D., PATRIZI, F., AND SARDIÑA, S. 2013. Automatic Behavior Composition Synthesis. *Artif. Intell.* 196, 106–142.
- GRÜNINGER, M. AND MENZEL, C. 2003. The Process Specification Language (PSL) Theory and Applications. *AI Magazine* 24, 3, 63–74.
- GU, Y. AND SOUTCHANSKI, M. 2007. Decidable Reasoning in a Modified Situation Calculus. In *IJCAI*, M. M. Veloso, Ed. AAAI Press/International Joint Conferences on Artificial Intelligence, 1891–1897.
- GUSTAFSSON, J. AND DOHERTY, P. 1996. Embracing Occlusion in Specifying the Indirect Effects of Actions. In *KR*. AAAI Press, 87–98.

- HATZI, O., VRAKAS, D., NIKOLAIDOU, M., BASSILIADES, N., ANAGNOSTOPOULOS, D., AND VLAHAVAS, I. P. 2012. An Integrated Approach to Automated Semantic Web Service Composition through Planning. *IEEE T. Services Computing* 5, 3, 319–332.
- HÖLLDOBLER, S. AND KUSKE, D. 2000. The Boundary between Decidable and Undecidable Fragments of the Fluent Calculus. In *LPAR 2000*, M. Parigot and A. Voronkov, Eds. LNCS Series, vol. 1955. Springer, 436–450.
- HOLZMANN, G. J. 2004. *The SPIN Model Checker - Primer and Reference Manual*. Addison-Wesley.
- HULL, R. AND SU, J. 2005. Tools for Composite Web Services: A Short Overview. *SIGMOD Record* 34, 2, 86–95.
- JAEGER, M. C., ROJEC-GOLDMANN, G., AND MÜHL, G. 2004. QoS Aggregation for Web Service Composition using Workflow Patterns. In *EDOC*. IEEE Computer Society, 149–159.
- JBOSS JBPM TEAM. 2013. jbpm business process management suite v6.0. Online (<http://jbpm.jboss.org>).
- KADNER, K., OBERLE, D., SCHAEFFLER, M., HORCH, A., KINTZ, M., BARTON, L., LEIDIG, T., PEDRINACI, C., DOMINGUE, J., ROMANELLI, M., TRAPERO, R., AND KUTSIKOS, K. 2011. Unified Service Description Language XG Final Report. World Wide Web Consortium, Incubator Group Report XGR-usdl-20111027.
- KAKAS, A. C., MICHAEL, L., AND MILLER, R. 2011. Modular-E and the role of elaboration tolerance in solving the qualification problem. *Artif. Intell.* 175, 1, 49–78.
- KAZHAMIKIN, R., PISTORE, M., AND SANTUARI, L. 2006. Analysis of Communication Models in Web Service Compositions. In *Proceedings of the 15th International Conference on World Wide Web*, L. Carr, D. D. Roure, A. Iyengar, C. A. Goble, and M. Dahlin, Eds. WWW '06. ACM, 267–276.
- KELLER, U. AND LAUSEN, H. 2006. Functional Description of Web Services. WSML Working Draft D28.1 v0.1, WSMO Working Group, January 2006.

- KLUSCH, M., FRIES, B., AND SYCARA, K. P. 2009. OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services. *J. Web Sem.* 7, 2, 121–133.
- KLUSCH, M. AND KAUFER, F. 2009. WSMO-MX: A hybrid Semantic Web service matchmaker. *Web Intelligence and Agent Systems* 7, 1, 23–42.
- KLUSCH, M., KHALID, M. A., KAPAHNKE, P., FRIES, B., AND VASILESKI, M. 2010. OWL-S TC: Service Retrieval Test Collection. Online (<http://projects.semwebcentral.org/projects/owls-tc/>).
- KOEHLER, J. AND SRIVASTAVA, B. 2003. Web Service Composition: Current Solutions and Open Problems. In *Proceedings of the ICAPS 2003 Workshop on Planning for Web Services*. AAAI Press, 28–35.
- KRITIKOS, K. AND PLEXOUSAKIS, D. 2009a. Mixed-Integer Programming for QoS-Based Web Service Matchmaking. *IEEE T. Services Computing* 2, 2, 122–139.
- KRITIKOS, K. AND PLEXOUSAKIS, D. 2009b. Requirements for QoS-based Web Service Description and Discovery. *IEEE T. Services Computing* 2, 4, 320–337.
- KÜSTER, U., STERN, M., AND KÖNIG-RIES, B. 2005. A Classification of Issues and Approaches in Automatic Service Composition. In *International Workshop on Engineering Service Compositions (WESC 05)*. IBM Research.
- KVARNSTRÖM, J. AND DOHERTY, P. 2000. Tackling the Qualification Problem Using Fluent Dependency Constraints. *Computational Intelligence* 16, 2, 169–209.
- LÉCUÉ, F., LÉGER, A., AND DELTEIL, A. 2008a. DL Reasoning and AI Planning for Web Service Composition. In *Web Intelligence*. IEEE, 445–453.
- LÉCUÉ, F., SILVA, E., AND PIRES FERREIRA, L. 2008b. A Framework for Dynamic Web Services Composition. In *Emerging Web Services Technology, Volume II*, T. Gschwind and C. Pautasso, Eds. Whitestein Series in Software Agent Technologies and Autonomic Computing. Birkhäuser Basel, 59–75.

- LEHMANN, H. AND LEUSCHEL, M. 2000. Decidability Results for the Propositional Fluent Calculus. In *Computational Logic — CL 2000*, J. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. Pereira, Y. Sagiv, and P. Stuckey, Eds. LNCS Series, vol. 1861. Springer, 762–776.
- LEMOS, F., GATER, A., GRIGORI, D., AND BOUZEGHOUB, M. 2012. A framework for service discovery based on structural similarity and quality satisfaction. In *ICWE*, M. Brambilla, T. Tokuda, and R. Tolksdorf, Eds. Lecture Notes in Computer Science Series, vol. 7387. Springer, 481–485.
- MABROUK, N. B., BEAUCHE, S., KUZNETSOVA, E., GEORGANTAS, N., AND ISSARNY, V. 2009. QoS-Aware Service Composition in Dynamic Service Oriented Environments. In *Middleware*, J. Bacon and B. F. Cooper, Eds. Lecture Notes in Computer Science Series, vol. 5896. Springer, 123–142.
- MAJITHIA, S., WALKER, D. W., AND GRAY, W. A. 2004. A Framework for Automated Service Composition in Service-Oriented Architectures. In *ESWS*, C. Bussler, J. Davies, D. Fensel, and R. Studer, Eds. Lecture Notes in Computer Science Series, vol. 3053. Springer, 269–283.
- MARCONI, A. AND PISTORE, M. 2009. Synthesis and Composition of Web Services. In *SFM*, M. Bernardo, L. Padovani, and G. Zavattaro, Eds. Lecture Notes in Computer Science Series, vol. 5569. Springer, 89–157.
- MARCONI, A., PISTORE, M., AND TRAVERSO, P. 2006. Specifying Data-Flow Requirements for the Automated Composition of Web Services. In *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006)*. IEEE, 147–156.
- MARCONI, A., PISTORE, M., AND TRAVERSO, P. 2008. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Eng. Bull.* 31, 3, 23–26.
- MARTIN, D., BURSTEIN, M., HOBBS, J., LASSILA, O., MCDERMOTT, D., MCILRAITH, S., NARAYANAN, S., PAOLUCCI, M., PARSIA, B., PAYNE, T., SIRIN, E., SRINI-



- VASAN, N., AND SYCARA, K. 2004. OWL-S: Semantic Markup for Web Services. <http://www.ai.sri.com/daml/services/owl-s/1.2/>.
- MCCARTHY, J. 1999. Elaboration tolerance. <http://www-formal.stanford.edu/jmc/elaboration/elaboration.html>.
- MCCARTHY, J. AND HAYES, P. J. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In *Machine Intelligence 4*, B. Meltzer and D. Michie, Eds. Edinburgh University Press, 463–502.
- MCILRAITH, S. A. AND SON, T. C. 2002. Adapting Golog for Composition of Semantic Web Services. In *KR, D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, Eds. Morgan Kaufmann*, 482–496.
- MELLO FERREIRA, A., KRITIKOS, K., AND PERNICI, B. 2009. Energy-Aware Design of Service-Based Applications. In *ICSOC-ServiceWave 2009*, L. Baresi, C.-H. Chi, and J. Suzuki, Eds. Lecture Notes in Computer Science Series, vol. 5900. Springer Berlin Heidelberg, 99–114.
- MESEGUER, P., ROSSI, F., AND SCHIEX, T. 2006. Chapter 9 - Soft Constraints. In *Handbook of Constraint Programming*, F. Rossi, P. van Beek, and T. Walsh, Eds. Foundations of Artificial Intelligence Series, vol. 2. Elsevier, 281 – 328.
- MILANOVIĆ, N. AND MALEK, M. 2004. Current Solutions for Web Service Composition. *IEEE Internet Computing* 8, 6, 51–59.
- MILICIC, M. 2008. Action, Time and Space in Description Logics. Ph.D. thesis, Technische Universität Dresden.
- MILLER, R. 2006. Three Problems in Logic-Based Knowledge Representation. *ASLIB Proceedings: New information perspectives* 58, 1/2, 140 – 151.
- MILNER, R. 2004. *Communicating and Mobile Systems: the Pi-Calculus* Fifth Ed. Cambridge University Press.

- OASIS. 2007. Web Services Business Process Execution Language Version 2.0. Online (<http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html>).
- OASIS. 2008. Reference Architecture for Service Oriented Architecture Version 1.0. Online (<http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html>).
- OBERLE, D., BARROS, A. P., KYLAU, U., AND HEINZL, S. 2013. A unified description language for human to automated services. *Inf. Syst.* 38, 1, 155–181.
- OBJECT MANAGEMENT GROUP. 2011. Business Process Model and Notation (BPMN). Online (<http://www.omg.org/spec/BPMN/>).
- OBJECT MANAGEMENT GROUP. 2012a. Common Object Request Broker Architecture (CORBA). Online (<http://www.omg.org/spec/CORBA/>).
- OBJECT MANAGEMENT GROUP. 2012b. Service Oriented Architecture Modeling Language (SoaML). Online (<http://www.omg.org/spec/SoaML/>).
- PACHOLSKI, L., SZWAST, W., AND TENDERA, L. 2000. Complexity Results for First-Order Two-Variable Logic with Counting. *SIAM J. Comput.* 29, 4, 1083–1117.
- PALIWAL, A. V., SHAFIQ, B., VAIDYA, J., XIONG, H., AND ADAM, N. R. 2012. Semantics-based automated service discovery. *IEEE T. Services Computing* 5, 2, 260–275.
- PAPAZOGLU, M. P., TRAVERSO, P., DUSTDAR, S., AND LEYMANN, F. 2007. Service-Oriented Computing: State of the Art and Research Challenges. *Computer* 40, 11, 38–45.
- PEDRINACI, C. AND DOMINGUE, J. 2010. Toward the Next Wave of Services: Linked Services for the Web of Data. *Journal of Universal Computer Science* 16, 13, 1694–1719.
- PINO, L. AND SPANOUDAKIS, G. 2012. Constructing Secure Service Compositions with Patterns. In *SERVICES*. IEEE, 184–191.

- PISTORE, M., BARBON, F., BERTOLI, P., SHAPARAU, D., AND TRAVERSO, P. 2004. Planning and Monitoring Web Service Composition. In *Artificial Intelligence: Methodology, Systems, and Applications*, C. Bussler and D. Fensel, Eds. Lecture Notes in Computer Science Series, vol. 3192. Springer Berlin Heidelberg, 106–115.
- PISTORE, M., MARCONI, A., BERTOLI, P., AND TRAVERSO, P. 2005a. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence. IJCAI'05*. Morgan Kaufmann Publishers Inc., 1252–1259.
- PISTORE, M., TRAVERSO, P., AND BERTOLI, P. 2005b. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 2005)*. AAAI Press, 2–11.
- RAN, S. 2003. A Model for Web Services Discovery With QoS. *SIGecom Exchanges* 4, 1, 1–10.
- RAO, J., DIMITROV, D., HOFMANN, P., AND SADEH, N. M. 2006. A mixed initiative approach to semantic web service discovery and composition: Sap's guided procedures framework. In *ICWS (2007-06-10)*. IEEE Computer Society, 401–410.
- RAO, J., KÜNGAS, P., AND MATSKIN, M. 2004. Logic-based Web Services Composition: From Service Description to Process Model. In *ICWS*. IEEE Computer Society, 446–453.
- RAO, J. AND SU, X. 2004. A Survey of Automated Web Service Composition Methods. In *SWSWPC*, J. Cardoso and A. P. Sheth, Eds. Lecture Notes in Computer Science Series, vol. 3387. Springer, 43–54.
- REITER, R. 1980. A Logic for Default Reasoning. *Artif. Intell.* 13, 1-2, 81–132.
- REITER, R. 1991. The Frame Problem in the Situation Calculus: A Simple Solution (Sometimes) and a Completeness Result for Goal Regression. In *Artificial Intelli-*

- gence and the Mathematical Theory of Computation: Papers in Honor of John McCarthy*. Academic Press, 359–380.
- RICHARDSON, L. AND RUBY, S. 2007. *RESTful Web Services*. O'Reilly.
- ROMAN, D., KELLER, U., LAUSEN, H., DE BRUIJN, J., LARA, R., STOLLBERG, M., POLLERES, A., FEIER, C., BUSSLER, C., AND FENSEL, D. 2005. Web Service Modeling Ontology. *Applied Ontology* 1, 77–106.
- ROSENBERG, F., CELIKOVIC, P., MICHLMAYR, A., LEITNER, P., AND DUSTDAR, S. 2009. An End-to-End Approach for QoS-Aware Service Composition. In *Proceedings of the 13th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2009)*. IEEE Computer Society, 151–160.
- RUSSELL, N., TER HOFSTEDE, A., VAN DER AALST, W., AND MULYAR, N. 2006. Workflow Control-Flow Patterns: A Revised View. Tech. Rep. BPM-06-22, BPM Center, June 2006.
- SCHIFFEL, S. AND THIELSCHER, M. 2006. Reconciling Situation Calculus and Fluent Calculus. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1. AAAI'06*. AAAI Press, 287–292.
- SHENG, Q. Z., MAAMAR, Z., YAO, L., SZABO, C., AND BOURNE, S. 2014. Behavior modeling and automated verification of Web services. *Inf. Sci.* 258, 416–433.
- SOHRABI, S. AND MCILRAITH, S. A. 2009. Optimizing Web Service Composition While Enforcing Regulations. In *ISWC 2009: Proceedings of the 8th International Semantic Web Conference, Chantilly, VA, USA*. Springer Berlin Heidelberg, 601–617.
- SOHRABI, S., PROKOSHYNA, N., AND MCILRAITH, S. A. 2009. Web Service Composition via the Customization of Golog Programs with User Preferences. In *Conceptual Modeling: Foundations and Applications*, A. Borgida, V. K. Chaudhri, P. Giorgini, and E. S. K. Yu, Eds. Lecture Notes in Computer Science Series, vol. 5600. Springer, 319–334.

- SPANOUKAKIS, G. AND ZISMAN, A. 2010. Discovering Services during Service-Based System Design Using UML. *IEEE Trans. Software Eng.* 36, 3, 371–389.
- TANG, X., JIANG, C., AND ZHOU, M. 2011. Automatic Web service composition based on Horn clauses and Petri nets. *Expert Syst. Appl.* 38, 10, 13024–13031.
- TERLOUW, L. AND ALBANI, A. 2013. An Enterprise Ontology-Based Approach to Service Specification. *Services Computing, IEEE Transactions on* 6, 1, 89–101.
- THIELSCHER, M. 1997. Ramification and Causality. *Artif. Intell.* 89, 1-2, 317–364.
- THIELSCHER, M. 1999. From Situation Calculus to Fluent Calculus: State Update Axioms as a Solution to the Inferential Frame Problem. *Artif. Intell.* 111, 1-2, 277–299.
- THIELSCHER, M. 2000. The Fluent Calculus. Tech. Rep. CL-2000-01, Dresden University of Technology, January 2000.
- THIELSCHER, M. 2001a. The Concurrent, Continuous Fluent Calculus. *Studia Logica* 67, 3, 315–331.
- THIELSCHER, M. 2001b. The Qualification Problem: A solution to the problem of anomalous models. *Artif. Intell.* 131, 1-2, 1–37.
- THIELSCHER, M. 2005a. FLUX: A Logic Programming Method for Reasoning Agents. *Theory and Practice of Logic Programming* 5, 533–565.
- THIELSCHER, M. 2005b. *Reasoning Robots*. Applied Logic Series, vol. 33. Springer Netherlands.
- TOSIC, V., PAGUREK, B., AND PATEL, K. 2003. WSOL - A Language for the Formal Specification of Classes of Service for Web Services. In *Proceedings of the International Conference on Web Services (ICWS 2003)*, L.-J. Zhang, Ed. CSREA Press, 375–381.

- TRAINOTTI, M., PISTORE, M., CALABRESE, G., ZACCO, G., LUCCHESI, G., BARBON, F., BERTOLI, P., AND TRAVERSO, P. 2005. ASTRO: Supporting Composition and Execution of Web Services. In *Service-Oriented Computing - ICSOC 2005*, B. Benatallah, F. Casati, and P. Traverso, Eds. Lecture Notes in Computer Science Series, vol. 3826. Springer Berlin Heidelberg, 495–501.
- TRAVERSO, P. AND PISTORE, M. 2004. Automated Composition of Semantic Web Services into Executable Processes. In *International Semantic Web Conference*. Springer Berlin Heidelberg, 380–394.
- TZENG, G. AND HUANG, J. 2011. *Multiple Attribute Decision Making: Methods and Applications*. Chapman and Hall/CRC.
- VAN DER AALST, W. AND TER HOFSTEDE, A. 2003. YAWL: Yet Another Workflow Language.
- VAN DER AALST, W., TER HOFSTEDE, A., KIEPUSZEWSKI, B., AND BARROS, A. 2003. Workflow Patterns. *Distributed and Parallel Databases* 14, 1, 5–51.
- VAQUERO, L. M., RODERO-MERINO, L., CACERES, J., AND LINDNER, M. 2009. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.* 39, 1, 50–55.
- WEI, Y. AND BLAKE, M. 2010. Service-Oriented Computing and Cloud Computing: Challenges and Opportunities. *Internet Computing, IEEE* 14, 6, 72–75.
- WSML WORKING GROUP. 2008a. The Web Service Modeling Language WSML.
- WSML WORKING GROUP. 2008b. WSML Abstract Syntax and Semantics. Tech. Rep. D16.3v1.0, WSMO Working Group, August 2008.
- WU, Z., RANABAHU, A., GOMADAM, K., SHETH, A. P., AND MILLER, J. A. 2007. Automatic Composition of Semantic Web Services using Process and Data Mediation. Tech. rep., Kno.e.sis Center, Wright State University, February 2007.

- ZEGINIS, C., KONSOLAKI, K., KRITIKOS, K., AND PLEXOUSAKIS, D. 2012. Towards Proactive Cross-Layer Service Adaptation. In *Web Information Systems Engineering - WISE 2012*, X. Wang, I. Cruz, A. Delis, and G. Huang, Eds. Lecture Notes in Computer Science Series, vol. 7651. Springer Berlin Heidelberg, 704–711.
- ZEGINIS, C., KRITIKOS, K., GAREFALAKIS, P., KONSOLAKI, K., MAGOUTIS, K., AND PLEXOUSAKIS, D. 2013. Towards Cross-Layer Monitoring of Multi-Cloud Service-Based Applications. In *Service-Oriented and Cloud Computing*, K.-K. Lau, W. Lamersdorf, and E. Pimentel, Eds. Lecture Notes in Computer Science Series, vol. 8135. Springer Berlin Heidelberg, 188–195.
- ZENG, L., NGU, A. H. H., BENATALLAH, B., PODOROZHNY, R. M., AND LEI, H. 2008. Dynamic composition and optimization of web services. *Distributed and Parallel Databases* 24, 1-3, 45–72.
- ZISMAN, A., MAHBUB, K., AND SPANOUDAKIS, G. 2007. A Service Discovery Framework based on Linear Composition. In *IEEE SCC (2007-07-10)*. IEEE Computer Society, 536–543.
- ZISMAN, A., SPANOUDAKIS, G., DOOLEY, J., AND SIVERONI, I. 2013. Proactive and reactive runtime service discovery: A framework and its evaluation. *IEEE Trans. Software Eng.* 39, 7, 954–974.





## Appendix A

### WSSL BNF grammar

$\langle wssl \rangle ::= \langle service \rangle \langle input \rangle^* \langle output \rangle^* \langle pre \rangle^* \langle post \rangle^* \langle causal \rangle^* \langle default \rangle? \langle quality \rangle^*$   
|  $\langle goal \rangle^+$

$\langle service \rangle ::= \text{'service' } \langle iri \rangle^+$

$\langle input \rangle ::= \text{'input' } \langle iri \rangle^+$

$\langle output \rangle ::= \text{'output' } \langle iri \rangle^+$

$\langle pre \rangle ::= \text{'precondition' } \langle iri \rangle \langle logexp \rangle$

$\langle post \rangle ::= \text{'postcondition' } \langle iri \rangle \langle updateexp \rangle$

$\langle causal \rangle ::= \text{'causalrelation' } \langle iri \rangle \langle updateexp \rangle$

$\langle default \rangle ::= \text{'defaulttheory' } \langle iri \rangle \langle observation \rangle^* \langle rule \rangle^+$

$\langle quality \rangle ::= \text{'qosprofile' } \langle iri \rangle \langle constraint \rangle^+$

$\langle goal \rangle ::= \text{'goal' } \langle iri \rangle \langle goalexp \rangle$

$\langle observation \rangle ::= \text{'observation' } \langle logexp \rangle$

$\langle rule \rangle ::= (\text{'prerequisite' } \langle logexp \rangle)? (\text{'justification' } \langle logexp \rangle)^+ \text{'conclusion' } \langle logexp \rangle$

$\langle constraint \rangle ::= \text{'constraint' } \langle logexp \rangle$

$\langle logexp \rangle ::= \langle term \rangle \langle comp\_op \rangle \langle term \rangle$   
|  $\text{'not' } \langle logexp \rangle$

|  $\langle \text{logexp} \rangle$  'and'  $\langle \text{logexp} \rangle$   
 |  $\langle \text{logexp} \rangle$  'or'  $\langle \text{logexp} \rangle$   
 |  $\langle \text{logexp} \rangle$  '->'  $\langle \text{logexp} \rangle$   
 |  $\langle \text{logexp} \rangle$  '(-)'  $\langle \text{logexp} \rangle$   
 | 'forall'  $\langle \text{var} \rangle$ +  $\langle \text{logexp} \rangle$   
 | 'exists'  $\langle \text{var} \rangle$ +  $\langle \text{logexp} \rangle$   
 | 'ramify'  $\langle \text{term} \rangle$   $\langle \text{term} \rangle$   $\langle \text{term} \rangle$   $\langle \text{term} \rangle$   
 |  $\langle \text{holds} \rangle$   
 |  $\langle \text{predicate} \rangle$   
 | 'true'  
 | 'false'

$\langle \text{updateexp} \rangle ::= \langle \text{logexp} \rangle$  'or'  $\langle \text{logexp} \rangle$

| 'exists'  $\langle \text{var} \rangle$ +  $\langle \text{logexp} \rangle$

$\langle \text{goalexp} \rangle ::= \langle \text{goalterm} \rangle$   $\langle \text{comp\_op} \rangle$   $\langle \text{goalterm} \rangle$

|  $\langle \text{goalexp} \rangle$  'and'  $\langle \text{goalexp} \rangle$

|  $\langle \text{goalexp} \rangle$  'or'  $\langle \text{goalexp} \rangle$

| 'ramify'  $\langle \text{goalterm} \rangle$   $\langle \text{goalterm} \rangle$   $\langle \text{goalterm} \rangle$   $\langle \text{goalterm} \rangle$

|  $\langle \text{goalholds} \rangle$

|  $\langle \text{goalpredicate} \rangle$

| 'true'

| 'false'

$\langle \text{holds} \rangle ::= \langle \text{name} \rangle$   $\langle \text{term} \rangle$

$\langle \text{goalholds} \rangle ::= \langle \text{name} \rangle$   $\langle \text{goalterm} \rangle$

$\langle \text{predicate} \rangle ::= \langle \text{iri} \rangle$   $\langle \text{terms} \rangle$

$\langle \text{goalpredicate} \rangle ::= \langle \text{iri} \rangle$   $\langle \text{goalterm} \rangle^*$

$\langle \text{term} \rangle ::= \langle \text{iri} \rangle$   $\langle \text{terms} \rangle$

|  $\langle \text{var} \rangle$

|  $\langle \text{term} \rangle$  'minus'  $\langle \text{term} \rangle$

---

|  $\langle term \rangle$  'plus'  $\langle term \rangle$   
|  $\langle term \rangle$  'o'  $\langle term \rangle$

$\langle terms \rangle ::= \langle term \rangle$   
| '('  $\langle terms \rangle$  ','  $\langle term \rangle$  ')'

$\langle goalterm \rangle ::= \langle iri \rangle \langle terms \rangle$   
|  $\langle var \rangle$

$\langle goalterms \rangle ::= \langle goalterm \rangle$   
| '('  $\langle goalterms \rangle$  ','  $\langle goalterm \rangle$  ')'

$\langle var \rangle ::= \langle name \rangle$

$\langle comp\_op \rangle ::= '>'$   
| '>='  
| '='  
| '!='

$\langle iri \rangle ::= \langle full\_iri \rangle$   
|  $\langle compact\_uri \rangle$

$\langle full\_iri \rangle ::= "$   $\langle iri\_f \rangle$   $"$

$\langle compact\_uri \rangle ::= (\langle name \rangle \#)? \langle name \rangle$

The definitions for  $\langle iri\_f \rangle$  and  $\langle name \rangle$  can be found in the WSML language reference [WSML Working Group 2008a].



## Appendix B

# WSSL XML Schema

### B.1 XML Schema

```
1 <?xml version="1.0"?>
2
3 <xs:schema version="1.0"
4     xmlns:xs="http://www.w3.org/2001/XMLSchema"
5     elementFormDefault="qualified">
6
7     <xs:simpleType name="wsslIRI">
8         <xs:union memberTypes="xs:anyURI"/>
9     </xs:simpleType>
10
11    <xs:element name="wssl">
12        <xs:complexType>
13            <xs:sequence>
14                <xs:choice>
15                    <xs:element name="service" type="wsslInOut"/>
16                    <xs:element name="goal" type="wsslGoalExp" minOccurs="1"
17                        maxOccurs="unbounded"/>
18                </xs:choice>
19                <xs:choice minOccurs="0" maxOccurs="unbounded">
20                    <xs:element name="input" type="wsslInOut"/>
21                    <xs:element name="output" type="wsslInOut"/>

```

```

22         <xs:element name="precondition" type="wsslLogExp"/>
23         <xs:element name="postcondition" type="wsslStateUpd"/>
24         <xs:element name="causal" type="wsslCausalRel"/>
25     </xs:choice>
26     <xs:element ref="defaultt" minOccurs="0" maxOccurs="1"/>
27     <xs:element ref="QoSProfile" minOccurs="0"
28         maxOccurs="unbounded"/>
29 </xs:sequence>
30 </xs:complexType>
31 </xs:element>
32
33 <xs:element name="defaultt">
34     <xs:complexType>
35         <xs:sequence>
36             <xs:element name="observation" type="wsslLogExp"
37                 minOccurs="0" maxOccurs="unbounded"/>
38             <xs:element ref="rule" minOccurs="1" maxOccurs="unbounded"/>
39         </xs:sequence>
40         <xs:attribute name="name" type="wsslIRI" />
41     </xs:complexType>
42 </xs:element>
43
44 <xs:element name="QoSProfile">
45     <xs:complexType>
46         <xs:sequence>
47             <xs:element name="constraint" type="wsslLogExp"/>
48         </xs:sequence>
49     </xs:complexType>
50 </xs:element>
51
52 <xs:element name="rule">
53     <xs:complexType>
54         <xs:sequence>
55             <xs:element name="prerequisite" type="wsslLogExp"
56                 minOccurs="0" maxOccurs="1"/>
57             <xs:element name="justification" type="wsslLogExp"
58                 minOccurs="1" maxOccurs="unbounded"/>

```

```
59         <xs:element name="conclusion" type="wsslLogExp"
60                   minOccurs="1" maxOccurs="1"/>
61     </xs:sequence>
62 </xs:complexType>
63 </xs:element>
64
65 <xs:group name="formula">
66     <xs:choice>
67         <xs:element name="true" type="emptyType"/>
68         <xs:element name="false" type="emptyType"/>
69         <xs:element ref="predicate"/>
70         <xs:element name="greaterThan" type="binaryTerm"/>
71         <xs:element name="greaterEqual" type="binaryTerm"/>
72         <xs:element name="equal" type="binaryTerm"/>
73         <xs:element name="notEqual" type="binaryTerm"/>
74         <xs:element name="lessEqual" type="binaryTerm"/>
75         <xs:element name="lessThan" type="binaryTerm"/>
76         <xs:element name="not" type="wsslLogExp"/>
77         <xs:element name="and" type="anyLogExp"/>
78         <xs:element name="or" type="anyLogExp"/>
79         <xs:element name="implies" type="binaryLogExp"/>
80         <xs:element name="equivalent" type="binaryLogExp"/>
81         <xs:element name="forall" type="quantifiedLogExp"/>
82         <xs:element name="exists" type="quantifiedLogExp"/>
83         <xs:element name="holds" type="stateful"/>
84         <xs:element name="ramify" type="quadTerm"/>
85     </xs:choice>
86 </xs:group>
87
88 <xs:group name="terms">
89     <xs:choice>
90         <xs:element name="term" type="wsslTerm"/>
91         <xs:element ref="var"/>
92         <xs:element name="empty" type="emptyType"/>
93         <xs:element name="minus" type="binaryTerm"/>
94         <xs:element name="plus" type="binaryTerm"/>
95         <xs:element name="circ" type="anyTerm"/>
```

```

96     </xs:choice>
97 </xs:group>
98
99 <xs:complexType name="wsslLogExp">
100     <xs:group ref="formula"/>
101     <xs:attribute name="name" type="wsslIRI"/>
102 </xs:complexType>
103
104 <xs:complexType name="wsslTerm" mixed="true">
105     <xs:group ref="terms" minOccurs="0" maxOccurs="unbounded"/>
106     <xs:attribute name="type" type="wsslIRI" use="required"/>
107 </xs:complexType>
108
109 <xs:complexType name="wsslInOut">
110     <xs:attribute name="name" type="wsslIRI"/>
111     <xs:attribute name="grounding" type="wsslIRI"/>
112 </xs:complexType>
113
114 <xs:complexType name="wsslStateUpd">
115     <xs:choice minOccurs="1" maxOccurs="1">
116         <xs:element name="exists" type="quantifiedLogExp"/>
117         <xs:element name="equal" type="binaryTerm"/>
118         <xs:element name="or" type="anyLogExp"/>
119         <xs:element name="ramify" type="quadTerm"/>
120     </xs:choice>
121     <xs:attribute name="name" type="wsslIRI"/>
122 </xs:complexType>
123
124 <xs:complexType name="wsslCausalRel">
125     <xs:choice minOccurs="1" maxOccurs="1">
126         <xs:element name="implies" type="binaryLogExp"/>
127     </xs:choice>
128     <xs:attribute name="name" type="wsslIRI"/>
129 </xs:complexType>
130
131 <xs:element name="predicate">
132     <xs:complexType>

```



```
133     <xs:sequence minOccurs="0" maxOccurs="unbounded">
134         <xs:group ref="terms"/>
135     </xs:sequence>
136     <xs:attribute name="name" type="wssLIURI" use="required"/>
137 </xs:complexType>
138 </xs:element>
139
140 <xs:complexType name="binaryTerm">
141     <xs:sequence minOccurs="2" maxOccurs="2">
142         <xs:group ref="terms"/>
143     </xs:sequence>
144 </xs:complexType>
145
146 <xs:complexType name="quadTerm">
147     <xs:sequence minOccurs="4" maxOccurs="4">
148         <xs:group ref="terms"/>
149     </xs:sequence>
150 </xs:complexType>
151
152 <xs:complexType name="anyTerm">
153     <xs:sequence minOccurs="2" maxOccurs="unbounded">
154         <xs:group ref="terms"/>
155     </xs:sequence>
156 </xs:complexType>
157
158 <xs:complexType name="binaryLogExp">
159     <xs:sequence minOccurs="2" maxOccurs="2">
160         <xs:group ref="formula"/>
161     </xs:sequence>
162 </xs:complexType>
163
164 <xs:complexType name="anyLogExp">
165     <xs:sequence minOccurs="2" maxOccurs="unbounded">
166         <xs:group ref="formula"/>
167     </xs:sequence>
168 </xs:complexType>
169
```

```
170 <xs:complexType name="quantifiedLogExp">
171   <xs:sequence>
172     <xs:element ref="var" minOccurs="1" maxOccurs="unbounded"/>
173     <xs:group ref="formula" minOccurs="1" maxOccurs="unbounded"/>
174   </xs:sequence>
175 </xs:complexType>
176
177 <xs:element name="var">
178   <xs:complexType>
179     <xs:attribute name="name" type="xs:string"/>
180     <xs:attribute name="value" type="xs:string"/>
181   </xs:complexType>
182 </xs:element>
183
184 <xs:complexType name="stateful">
185   <xs:sequence>
186     <xs:group ref="terms" minOccurs="1" maxOccurs="1"/>
187   </xs:sequence>
188   <xs:attribute name="state" type="xs:string" use="required"/>
189 </xs:complexType>
190
191 <xs:complexType name="emptyType">
192   <xs:complexContent>
193     <xs:restriction base="xs:anyType"/>
194   </xs:complexContent>
195 </xs:complexType>
196
197
198
199
200 <xs:complexType name="wsslGoalExp">
201   <xs:group ref="goalFormula"/>
202   <xs:attribute name="name" type="wsslIRI"/>
203 </xs:complexType>
204
205 <xs:group name="goalFormula">
206   <xs:choice>
```

```
207     <xs:element name="true" type="emptyType"/>
208     <xs:element name="false" type="emptyType"/>
209     <xs:element name="predicate" type="predicateGoal"/>
210     <xs:element name="and" type="anyGoalExp"/>
211     <xs:element name="or" type="anyGoalExp"/>
212     <xs:element name="holds" type="statefulGoal"/>
213     <xs:element name="ramify" type="quadGoalTerm"/>
214     <xs:element name="greaterThan" type="binaryTerm"/>
215     <xs:element name="greaterEqual" type="binaryTerm"/>
216     <xs:element name="equal" type="binaryTerm"/>
217     <xs:element name="notEqual" type="binaryTerm"/>
218     <xs:element name="lessEqual" type="binaryTerm"/>
219     <xs:element name="lessThan" type="binaryTerm"/>
220   </xs:choice>
221 </xs:group>
222
223 <xs:group name="goalTerms">
224   <xs:choice>
225     <xs:element name="term" type="wsslTerm"/>
226     <xs:element ref="var"/>
227   </xs:choice>
228 </xs:group>
229
230 <xs:complexType name="predicateGoal">
231   <xs:sequence minOccurs="0" maxOccurs="unbounded">
232     <xs:group ref="goalTerms"/>
233   </xs:sequence>
234   <xs:attribute name="name" type="wsslIRI" use="required"/>
235 </xs:complexType>
236
237 <xs:complexType name="anyGoalExp">
238   <xs:sequence minOccurs="2" maxOccurs="unbounded">
239     <xs:group ref="goalFormula"/>
240   </xs:sequence>
241 </xs:complexType>
242
243 <xs:complexType name="statefulGoal">
```

```

244     <xs:sequence>
245         <xs:group ref="goalTerms" minOccurs="1" maxOccurs="1"/>
246     </xs:sequence>
247     <xs:attribute name="state" type="xs:string" use="required"/>
248 </xs:complexType>
249
250 <xs:complexType name="quadGoalTerm">
251     <xs:sequence minOccurs="4" maxOccurs="4">
252         <xs:group ref="goalTerms"/>
253     </xs:sequence>
254 </xs:complexType>
255
256 </xs:schema>

```

## B.2 XML Encoding of running example tasks

### B.2.1 ReceiveCall

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <wssl
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xmlns:owlq="OWL-Q_Attribute.owl"
6     xsi:noNamespaceSchemaLocation="wssl4.xsd"
7     xmlns:wssl="http://www.example.org/wssl">
8
9     <service name="wssl#ReceiveCall"/>
10
11     <input name="wssl#call"/>
12
13     <output name="wssl#request"/>
14
15     <precondition name="wssl#PrecRC">
16         <holds state="?z_in">
17             <term type="wssl#CallCenterUp"/>

```

```
18     </holds>
19 </precondition>
20
21 <postcondition name="wssl#PostRC">
22   <equal>
23     <var name="?z_out"/>
24     <plus>
25       <var name="?z_in"/>
26       <term type="wssl#Received">
27         <var name="request"/>
28         <var name="sms"/>
29       </term>
30     </plus>
31   </equal>
32 </postcondition>
33
34 <QoSProfile>
35   <constraint>
36     <greaterEqual>
37       <term type="owlq:Availability"/>
38       <term type="xs:decimal">0.99</term>
39     </greaterEqual>
40   </constraint>
41 </QoSProfile>
42
43 </wssl>
```

## B.2.2 MReport

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <wssl
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:noNamespaceSchemaLocation="wssl4.xsd"
6   xmlns:wssl="http://www.example.org/wssl">
7
```

```
8 <service name="wssl#RetrieveDiag"/>
9
10 <input name="wssl#request"/>
11
12 <output name="wssl#status"/>
13
14 <precondition name="wssl#PrecRD">
15   <holds state="?z_in">
16     <term type="wssl#SystemActive">
17       <var name="vehicle"/>
18     </term>
19   </holds>
20 </precondition>
21
22 <postcondition name="wssl#PostRD">
23   <ramify>
24     <var name="?z_in"/>
25     <term type="wssl#Retrieved">
26       <var name="status"/>
27       <var name="vehicle"/>
28     </term>
29     <empty/>
30     <var name="?z_out"/>
31   </ramify>
32 </postcondition>
33
34 <causal name="wssl#CausalRD">
35   <implies>
36     <holds state="?p">
37       <term type="wssl#Retrieved">
38         <var name="status"/>
39         <var name="vehicle"/>
40       </term>
41     </holds>
42     <predicate name="causes">
43       <var name="?z"/>
44       <var name="?p"/>
```

```
45         <var name="?n"/>
46         <plus>
47             <var name="?z"/>
48             <term type="wssl#Generated">
49                 <var name="mechlog"/>
50             </term>
51         </plus>
52         <plus>
53             <var name="?p"/>
54             <term type="wssl#Generated">
55                 <var name="mechlog"/>
56             </term>
57         </plus>
58         <var name="?n"/>
59         <var name="?s"/>
60     </predicate>
61 </implies>
62 </causal>
63
64 </wssl>
```

### B.2.3 RetrieveDiag

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <wssl
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:noNamespaceSchemaLocation="wssl4.xsd"
6     xmlns:wssl="http://www.example.org/wssl">
7
8     <service name="wssl#MReport">
9     </service>
10
11     <input name="wssl#invoice"/>
12
13     <output name="wssl#report"/>
```

```
14
15 <precondition name="wssl#PrecMR">
16   <and>
17     <holds state="?z_in">
18       <term type="wssl#PayCompleted">
19         <var name="payform"/>
20       </term>
21     </holds>
22     <holds state="?z_in">
23       <term type="wssl#Generated">
24         <var name="mechlog"/>
25       </term>
26     </holds>
27     <not>
28       <holds state="?z_in">
29         <term type="wssl#Delivered">
30           <var name="report"/>
31         </term>
32       </holds>
33     </not>
34   </and>
35 </precondition>
36
37 <postcondition name="wssl#PostMR">
38   <or>
39     <equal>
40       <var name="?z_out"/>
41     <plus>
42       <var name="?z_in"/>
43         <term type="wssl#Delivered">
44           <var name="report"/>
45         </term>
46       </plus>
47     </equal>
48   <exists>
49     <var name="?deliv"/>
50   <predicate name="Acc">
```



```
51         <term type="Failure">
52             <var name="?deliv"/>
53         </term>
54         <var name="?z_in"/>
55     </predicate>
56         <equal>
57             <var name="?z_out"/>
58             <var name="?z_in"/>
59         </equal>
60 </exists>
61 </or>
62 </postcondition>
63
64 <causal name="wssl#CausalRD">
65     <implies>
66         <term type="wssl#Retrieved">
67             <var name="status"/>
68             <var name="vehicle"/>
69         </term>
70         <term type="wssl#Generated">
71             <var name="mechlog"/>
72         </term>
73     </implies>
74 </causal>
75
76 <defaultt>
77     <rule>
78         <justification>
79             <not>
80                 <predicate name="Acc">
81                     <var name="?c"/>
82                     <var name="?s_in"/>
83                 </predicate>
84             </not>
85         </justification>
86         <conclusion>
87             <not>
```

```
88         <predicate name="Acc">
89             <var name="?c"/>
90             <var name="?s_in"/>
91         </predicate>
92     </not>
93 </conclusion>
94 </rule>
95 </defaultt>
96
97 </wssl>
```

## Appendix C

# FLUX Kernels

### C.1 Copyright Notice

The code below is a modification of the original FLUX kernel, which can be found online in the address <http://www.flux-agent.org> and is associated with the following copyright and permission notice:

Copyright (c) 2000 by The Dresden University of Technology, Dresden, Saxony, Germany. All Rights Reserved.

Permission to use, copy, and modify, this software and its documentation for non-commercial research purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both the copyright notice and this permission notice appear in supporting documentation, and that the name of The Dresden University of Technology not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. The Dresden University of Technology makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

## C.2 Basic Kernel

Note that CHR code is not included. The basic kernel only requires the definition of `not_holds`, while the full kernel requires the complete CHR set of the original FLUX kernel.

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %%
3 %% Libraries
4 %%
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
7 %%
8 %% constraint handling rules (Eclipse library)
9 %%
10 :- lib (ech).
11
12 %%
13 %% FLUX constraint handling rule for not_holds
14 %%
15 :- chr (' notholds.chr ').
16
17
18 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
19 %%
20 %% State Specifications and Update
21 %%
22 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
23
24 %%
25 %% holds(F,Z): asserts that fluent F holds in state Z
26 %%
27 holds(F, [F|_]).
28 holds(F, Z) :- Z=[F1|Z1], F\==F1, holds(F, Z1).

```

```

29
30
31 %%
32 %% holds(F,Z,Zp): asserts that fluent F holds in state Z
33 %% state Zp is Z without F.
34 %%
35 holds(F, [F|Z], Z).
36 holds(F, Z, [F1|Zp]) :- Z=[F1|Z1], F\==F1, holds(F, Z1, Zp).
37
38
39 %%
40 %% minus(Z1,ThetaN,Z2): state Z2 is state Z1 minus the fluents in list ThetaN
41 %%
42 minus_(Z, [], Z).
43 minus_(Z, [F|Fs], Zp) :-
44     (\+ holds(F, Z)      -> Z1 = Z ;
45     holds(F, Z, Z1) ), minus_(Z1, Fs, Zp).
46
47
48 %%
49 %% plus(Z1,ThetaP,Z2): state Z2 is state Z1 plus the fluents in list ThetaP
50 %%
51 %% In case a fluent represents an output, the equivalent hasinput fluent
52 %% is added, in order to facilitate data flow routing for composition
53 %%
54 plus_(Z, [], Z).
55 plus_(Z, [F|Fs], Zp) :-
56     ( \+ holds(F, Z)      -> Z1=[F|Z],
57     (F=hasoutput(N) -> Z2=[hasinput(N)|Z1] ; Z2=Z1) ;
58     holds(F, Z), Z2=Z), plus_(Z2, Fs, Zp).
59
60
61 %%
62 %% update(Z1,ThetaP,ThetaN,Z2): state Z2 is state Z1 minus the fluents in

```





### C.3 Full Kernel

```

1 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2 %%
3 %% Libraries
4 %%
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6
7 %%
8 %% finite domain constraint solver (Eclipse library)
9 %%
10 :- lib(fd).
11
12 %%
13 %% constraint handling rules (Eclipse library)
14 %%
15 %:- lib(chr).
16 :- lib(ech).
17
18 %%
19 %% FLUX constraint handling rules
20 %%
21 :- chr(' fluent .chr ').
22
23
24 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
25 %%
26 %% State Specifications and Update
27 %%
28 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
29
30 %%
31 %% holds(F,Z): asserts that fluent F holds in state Z
32 %%

```



```

33 holds(F, [F|_]).
34 holds(F, Z) :- nonvar(Z), Z=[F1|Z1], F\==F1, holds(F, Z1).
35
36
37 %%
38 %% holds(F,Z,Zp): asserts that fluent F holds in state Z
39 %% state Zp is Z without F.
40 %%
41 holds(F, [F|Z], Z).
42 holds(F, Z, [F1|Zp]) :- nonvar(Z), Z=[F1|Z1], F\==F1, holds(F, Z1, Zp).
43
44
45 %%
46 %% cancel(F,Z1,Z2): state Z2 is state Z1 with all
47 %% (positive , negative , disjunctive ) knowledge of fluent F canceled
48 %%
49 cancel(F,Z1,Z2) :-
50     var(Z1)    -> cancel(F,Z1), cancelled(F,Z1), Z2=Z1 ;
51     Z1 = [G|Z] -> ( F\=G -> cancel(F,Z,Z3), Z2=[G|Z3]
52                 ; cancel(F,Z,Z2) ) ;
53     Z1 = []    -> Z2 = [].
54
55
56 %%
57 %% minus(Z1,ThetaN,Z2): state Z2 is state Z1 minus the fluents in list ThetaN
58 %%
59 minus_(Z, [], Z).
60 minus_(Z, [F|Fs], Zp) :-
61     ( \+ not_holds(F, Z) -> holds(F, Z, Z1) ;
62     \+ holds(F, Z)      -> Z1 = Z
63     ; cancel(F, Z, Z1), not_holds(F, Z1) ),
64     minus_(Z1, Fs, Zp).
65
66

```

```

67 %%
68 %% plus(Z1,ThetaP,Z2): state Z2 is state Z1 plus the fluents in list ThetaP
69 %%
70 %% In case a fluent represents an output, the equivalent hasinput fluent
71 %% is added, in order to facilitate data flow routing for composition
72 %%
73 plus_(Z, [], Z).
74 plus_(Z, [F|Fs], Zp) :-
75     ( \+ holds(F, Z) -> Z1=[F|Z],
76       (F=hasoutput(N) -> Z2=[hasinput(N)|Z1] ; Z2=Z1) ;
77       \+ not_holds(F, Z) -> Z2=Z
78     ; cancel(F, Z, Z3), not_holds(F, Z3), Z2=[F|Z3] ),
79     plus_(Z2, Fs, Zp).
80
81
82 %%
83 %% update(Z1,ThetaP,ThetaN,Z2): state Z2 is state Z1 minus the fluents in
84 %% list ThetaN plus the fluents in list ThetaP
85 %%
86 update(Z1, ThetaP, ThetaN, Z2) :-
87     minus_(Z1, ThetaN, Z), plus_(Z, ThetaP, Z2).
88
89
90 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
91 %%
92 %% State Knowledge
93 %%
94 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
95
96 %%
97 %% knows(F,Z): ground fluent F is known to hold in state Z
98 %%
99 knows(F, Z) :- \+ not_holds(F, Z).
100

```



```

135 %% Service Composition
136 %%
137 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
138
139 poss_if(F, A, B, Z) :- holds(F,Z), poss(A, Z) ;
140                        not_holds(F, Z), poss(B, Z), A \== B.
141 poss_and(A, B, Z) :- poss(A, Z), poss(B, Z), A \== B, A @< B.
142 poss_or(A, B, Z) :- poss(A, Z), poss(B, Z), A \== B, A @< B.
143 poss_xor(A, B, Z) :- poss(A, Z), poss(B, Z), A \== B, A @< B.
144 poss_loop(F,K,A,Z) :- K\==0, (holds(F,Z) -->
145   poss(A,Z)), update(Z,A,Z_PR), poss_loop(F,K-1,A,Z_PR).
146
147 state_update_if(Z, F, A, B, Z_PR) :-
148   holds(F, Z), state_update(Z, A, Z_PR) ;
149   not_holds(F, Z), state_update(Z, B, Z_PR).
150 state_update_and(Z, A, B, Z_PR) :- state_update(Z, A, Z_1),
151   state_update(Z_1, B, Z_PR).
152 state_update_or(Z,A,B,Z_PR) :- state_update(Z, A, Z_1),
153   state_update(Z_1, B, Z_PR) ;
154   state_update(Z,A,Z_PR) ;
155   state_update(Z,B,Z_PR).
156 state_update_xor(Z, A, B, Z_PR) :- state_update(Z, A, Z_PR),
157   \+ state_update(Z, B, Z_PR) ;
158   state_update(Z, A, Z_PR),
159   \+ state_update(Z, B, Z_PR).
160 state_update_loop(Z,F,K,A,Z_PR) :- not_holds(F,Z) --> Z_PR=Z ; K\==0,
161   (holds(F,Z), update(Z,A,Z_1),
162   not_holds(F,Z_1)) -->
163   state_update(Z,A,Z_1),
164   state_update_loop(Z_1,F,K-1,A,Z_PR).

```

## C.4 FLUX code of the running example

```

1  init(Z0) :- Z0 = [callcenterup, hasinput(call)].
2
3  poss(receivesms, Z) :- holds(hasinput(sms), Z), holds(callcenterup, Z).
4  poss(receivecall, Z) :- holds(hasinput(call), Z), holds(callcenterup, Z).
5  poss(retrievelocation, Z) :- holds(hasinput(request), Z),
6                               holds(gpsactive(user), Z).
7  poss(retrievediagnostics, Z) :- holds(hasinput(request), Z),
8                               holds(systemactive(vehicle), Z).
9  poss(findmech, Z) :- holds(hasinput(status), Z), holds(hasinput(location), Z),
10                      holds(retrieved(location, user), Z), holds(retrieved(status, vehicle), Z),
11                      not_holds(solved(status, location), Z).
12  poss(receivepay(payform, invoice), Z) :- holds(hasinput(payform), Z),
13                                           holds(solved(status, location), Z).
14  poss(ereport(invoice, report), Z) :- holds(hasinput(invoice), Z),
15                                         holds(paycompleted(payform), Z), holds(generated(mechlog), Z),
16                                         not_holds(emailed(report), Z).
17  poss(mreport(invoice, report), Z) :- holds(hasinput(invoice), Z),
18                                         holds(paycompleted(payform), Z), holds(generated(mechlog), Z),
19                                         not_holds(delivered(report), Z).
20
21  state_update(Z, receivesms, Z_PR) :- update(Z, [hasoutput(request),
22                                           received(request, sms)], [hasinput(sms)], Z_PR).
23  state_update(Z, receivecall, Z_PR) :- update(Z, [hasoutput(request),
24                                           received(request, call)], [hasinput(call)], Z_PR).
25  state_update(Z, retrievelocation, Z_PR) :- update(Z, [hasoutput(location),
26                                           received(location, user)], [hasinput(request)], Z_PR).
27  state_update(Z, retrievediagnostics, Z_PR) :- ramify(Z, [hasoutput(status),
28                                           received(status, vehicle)], [hasinput(request)], Z_PR).
29  state_update(Z, findmech, Z_PR) :- update(Z, [hasoutput(payform),
30                                           solved(status, location)], [hasinput(status), hasinput(location)], Z_PR).
31  state_update(Z, receivepay, Z_PR) :- ramify(Z, [hasoutput(invoice),
32                                           paycompleted(payform)], [hasinput(payform)], Z_PR).

```

```

33 state_update(Z, ereport, Z_PR) :- update(Z, [hasoutput(report), emailed(report)],
34                                     [hasinput(invoice)], Z_PR).
35 state_update(Z, mreport, Z_PR) :- update(Z,[hasoutput(report), delivered(report)],
36                                     [hasinput(invoice)], Z_PR)
37                                     ; update(Z, [failure(deliv)], [], Z_PR).
38
39 causes(Z, P, N, [generated(mechlog)|Z], _, _) :-
40     P = [hasoutput(status), received(status, vehicle)], N = [hasinput(request)].
41 causes(Z, P, N, [deactivated(creditcard)|Z], _, _) :-
42     holds(dailylimitreached(payform, creditcard), Z),
43     P = [hasoutput(invoice), paycompleted(payform)], N = [hasinput(payform)].
44
45 assist_plan(Z, [A|P], Z_PR) :- A_1 = receivesms, A_2 = receivecall,
46                               A = xor(A_1, A_2), poss_xor(A_1, A_2, Z),
47                               state_update_xor(Z, A_1, A_2, Z_1),
48                               assist_plan1(Z_1, P, Z_PR).
49 assist_plan1(Z, [A|P], Z_PR) :- A_1 = retrievelocation, A_2 = retrievediagnostics,
50                               A = and(A_1, A_2), poss_and(A_1, A_2, Z),
51                               state_update_and(Z, A_1, A_2, Z_1),
52                               assist_plan2(Z_1, P, Z_PR).
53 assist_plan2(Z, [A|P], Z_PR) :- A = findmech, poss(A, Z),
54                               state_update(Z, A, Z_1),
55                               assist_plan3(Z_1, P, Z_PR).
56 assist_plan3(Z, [A|P], Z_PR) :- A = receivepay, poss(A, Z),
57                               state_update(Z, A, Z_1),
58                               assist_plan4(Z_1, P, Z_PR).
59 assist_plan4(Z, A, Z_PR) :- F = req_deliv(report), A_1 = ereport, A_2 = mreport,
60                               A = if(F, A_1, A_2), poss_if(F, A_1, A_2, Z),
61                               state_update_if(Z, F, A_1, A_2, Z_PR).

```