University of Crete
Computer Science Department

Web service composition based on
enhanced specifications

Vassiliki Alevizou
Master’s Thesis

Heraklion, July 2005
ΣΥΝΘΕΣΗ ΗΛΕΚΤΡΟΝΙΚΩΝ ΥΠΗΡΕΣΙΩΝ ΒΑΣΙΣΜΕΝΗ ΣΕ ΕΜΠΛΟΥΤΙΣΜΕΝΕΣ ΠΡΟΔΙΑΓΡΑΦΕΣ

Εργασία που υποβλήθηκε από την
Αλεβίζου Βασιλική
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Αλεβίζου Βασιλική
Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Δημήτρης Πλεξουσάκης, Αναπλ. Καθηγητής, Επόπτης

Βασίλης Χριστοφίδης, Αναπλ. Καθηγητής, Μέλος

Κώστας Στεφανίδης, Καθηγητής, Μέλος

Δεκτή:

Δημήτρης Πλεξουσάκης, Αναπληρωτής Καθηγητής,
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Ιούλιος 2005
Web service composition based on enhanced specifications

Vassiliki Alevizou

Master's Thesis

Computer Science Department, University of Crete

Abstract

In recent years, the service-oriented computing research community has paid a lot of attention to Web service specification, discovery and composition. Web services can be seen as applications accessible to other applications over the Web. As Web services become commodities and their numbers increase, more efficient platforms supporting their orchestration and deployment in applications are needed. The ability to efficiently select and compose services on the Web in order to accomplish a complex user goal is of utmost importance. However, current Web service descriptions do not enable the possible selection and composition of available services. Several proposals for addressing the Web service composition problem, which can also be seen as a workflow or an AI planning problem, have appeared in the literature. However, service composition still lacks satisfactory solutions.

Current Web service specifications provide services with the definition of preconditions and postconditions, i.e., conditions that must be satisfied in the initial and final state of the service. However, these types of conditions are not sufficient for Web service composition. Due to the need for composition of a set of services, several conditions and constraints that did not exist in the individual service specifications arise. For the definition of these conditions, we augment Web service specifications with invariant assertions. These assertions must be satisfied throughout the set of states of a complex service. In addition, a set of composition rules is proposed. Given specific preconditions, postconditions and invariants of the Web services that need to be composed, we use these rules to determine the effects of the complex Web service in the presence of complex operations.

Composition rules enable reasoning about service composition and service interactions. They can be used to provide compositionality knowledge in composite service descriptions. The consideration of such rules in planning - especially in backward planning - can provide more effectiveness patterns for service selection as compared to
the mere matching between inputs / outputs and preconditions / postconditions. These patterns are based on rule satisfiability proving, and provide the generation of a plan, the execution of which will produce the desired goal.

Supervisor: Dimitris Plexousakis
Associate Professor
Σύνθεση ηλεκτρονικών υπηρεσιών βασισμένη σε εμπλουτισμένες προδιαγραφές

Αλεβίζου Βασιλική

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών, Πανεπιστήμιο Κρήτης

Περίληψη

Τα τελευταία χρόνια, η ερευνητική κοινότητα που ασχολείται με τις υπηρεσίες έχει δώσει ιδιαίτερη προσοχή στην περιγραφή, αναλύση και σύνθεση των ηλεκτρονικών υπηρεσιών. Οι ηλεκτρονικές υπηρεσίες μπορούν να χαρακτηριστούν σαν εφαρμογές που είναι προσβάσιμες από άλλες εφαρμογές πάνω από το δίκτυο. Καθώς οι ηλεκτρονικές υπηρεσίες τέλος να γίνουν εμπορεύσιμα αίδη και ο αριθμός τους αυξήθηκε, η ανάγκη για περισσότερες αποτελεσματικές πλατφόρμες οι οποίες υποστηρίζουν τον συντονισμό και την ανάπτυξη των εφαρμογών, αυξήθηκε. Η ικανότητα αποδοτικής επιλογής και σύνθεσης υπηρεσιών για την ικανοποίηση ενός σύνθετου στόχου του χρήστη, είναι ύψιστης σημασίας. Εν τούτω, οι εξέλιξες περιγραφής των ηλεκτρονικών υπηρεσιών δεν ενισχύουν την αποδοτική επιλογή και σύνθεση των υπάρχουσων υπηρεσιών. Πολλές δουλειές έχουν προσταθεί, μέχρι στιγμής, για την παρουσίαση του προβλήματος της σύνθεσης των ηλεκτρονικών υπηρεσιών, το οποίο μπορεί να παρουσιαστεί σαν ένα πρόβλημα ροών εργασιών ή προγραμματισμού ενεργειών. Αν και πολλές τεχνικές έχουν προσταθεί για την αντιμετώπιση αυτού του προβλήματος, η σύνθεση των υπηρεσιών στερείται ακόμα από ικανοποιητικές λύσεις.

Οι υπάρχουσες προδιαγραφές των ηλεκτρονικών υπηρεσιών προσφέρουν στις υπηρεσίες τον ορισμό προσωπικών και μετασωμικών, για παράδειγμα, συνήθως που θα πρέπει να υσχύουν στην αρχή και τελική κατάσταση της υπηρεσίας. Ωστόσο, αυτού του σιδους οι συνήθες δεν είναι ακριβείς για την σύνθεση ηλεκτρονικών υπηρεσιών. Εξαιτίας της ανάγκης για σύνθεση ενός συνόλου από υπηρεσίες, προοπτικά πολλές συνήθεις και περιορισμοι που δεν ορίζονται στις ατομικές προδιαγραφές των υπηρεσιών. Για τον ορισμό αυτών των συνθηκών, εμπλουτίσαμε τις προδιαγραφές των ηλεκτρονικών υπηρεσιών με αμετάβλητες δηλώσεις. Λοιπόν οι δηλώσεις θα πρέπει να ικανοποιούνται από όλες τις
ενδιάμεσες καταστάσεις της σύνθετης υπηρεσίας. Επιπλέον, ένα σύνολο από κανόνες σύνθεσης, προτάθηκε. Δεδομένων των ακριβών προσυνθηκών, μετασυνθηκών και περιορισμών των ηλεκτρονικών υπηρεσιών που χρειάζονται να συντεθούν, μπορούμε να χρησιμοποιήσουμε τους κανόνες για τον καθορισμό των αποτελεσμάτων των σύνθετων ηλεκτρονικών υπηρεσιών παρουσία σύνθετων λειτουργιών.

Οι κανόνες σύνθεσης ενδυναμώνουν τον συλλογισμό για την σύνθεση και την αλληλεπίδραση των υπηρεσιών. Μπορούμε να χρησιμοποιήσουμε για την παροχή γνώσης συνθετότητας στις περιγραφές των σύνθετων υπηρεσιών. Η θεώρηση τέτοιων κανόνων στο προγραμματισμό ενέργειων (planning) – ειδικά στον αντίστροφο προγραμματισμό – παρέχει αποδοτικότερες τρόπους επιλογής υπηρεσιών σε σύγκριση με τις απλές μεθόδους συσχετισμού μεταξύ εισόδων / εξόδων και προσυνθηκών / μετασυνθηκών. Οι συγκεκριμένες μέθοδοι επιλογής βασίζονται στην απόδειξη της ικανοποιησιμότητας των κανόνων και παρέχουν την δημιουργία ενός σχεδίου, η εκτέλεσή του οποίου θα μας προσφέρει το επιθυμητό αποτέλεσμα.

Επόπτης Καθηγητής: Δημήτρης Πλέξουσάκης
Αναπληρωτής Καθηγητής
Στους γονείς μου και
στην αδελφή μου Ελένη
Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω τον επόπτη μου, κ. Δημήτρη Πλαζουσάκη, για την σημαντική καθοδήγησή και την πολύτιμη βοήθειά του. Σημαντικοί παράγοντες για την εκπόνηση της συγκεκριμένης εργασίας αποτέλεσαν το ενδιαφέρον και η κατανόηση του τα δύο χρόνια των μεταπτυχιακών μου σπουδών.

Θα ήθελα να ευχαριστήσω τον κ. Βασίλη Χριστοφίδη για την συνεργασία που είχαμε αλλά και για τις πολύτιμες παρατηρήσεις και καθοδήγηση για τους πρώτους μήνες των μεταπτυχιακών μου σπουδών.

Ακόμα, θα ήθελα να ευχαριστήσω και τον κ. Κώστα Στεφανίδη, μέλος της εξεταστικής επιτροπής της μεταπτυχιακής μου εργασίας, για τις πολύτιμες παρατηρήσεις του.

Επίσης, θα ήθελα να ευχαριστήσω το Πανεπιστήμιο Κρήτης και την ομάδα των Πληροφορικών Συστημάτων για τα όσα μου πρόσφεραν κατά την διάρκεια των σπουδών μου στην Κρήτη.

Ένα μεγάλο ευχαριστώ αξιζεί στους φίλους μου και συγκεκριμένα στους (Γιώργο, Γιώργο, Δέσποινα, Ιωάννα, Νίκο και Παναγιώτη) για την συμπαράστασή και την κατανόηση που έδειχναν όλα αυτά τα χρόνια.

Το μεγαλύτερο ευχαριστώ όμως το αξίζουν οι γονείς μου, Διονύσης, και Μάρθα, καθώς και η αδελφή μου, Ελένη, που όλα αυτά τα χρόνια ήταν δίπλα μου, που πιστεύαν στις προσπάθειες μου και με στήριζαν ηθικά αλλά και υλικά. Σας ευχαριστώ πολύ για όλα!

Βασιλική Αλεβίζου
Contents

Contents ..........................................................................................................................................................i
List of Figures ...............................................................................................................................................v
Chapter 1 ....................................................................................................................................................1
Introduction ..................................................................................................................................................1
1.1 Motivation ........................................................................................................................................1
1.2 Contributions .....................................................................................................................................3
1.3 Organization of the document .........................................................................................................4
Chapter 2 ....................................................................................................................................................7
Web Service Composition: State of the Art ...............................................................................................7
  2.1 Web Services Standards ....................................................................................................................8
    2.1.1 Specification languages for Semantic Web services ...............................................................11
      2.1.1.1 Semantic Web Service Description with OWL-S ......................................................11
      2.1.1.2 Web Service Modelling Ontology .............................................................................15
      2.1.1.3 Comparing OWL-S and WSMO ..............................................................................18
    2.2 Defining Rules in the domain of the Semantic Web .................................................................19
    2.3 Web Service Composition ............................................................................................................23
      2.3.1 Web Services Composition Framework ............................................................................25
      2.3.2 Workflow problem ..............................................................................................................27
      2.3.3 Planning problem ................................................................................................................28
        2.3.3.1 Planning Domain Definition Language (PDDL) ..................................................30
        2.3.3.2 Hierarchical Task Network Planning ......................................................................31
        2.3.3.3 High-level Program Execution ...............................................................................35
        2.3.3.4 Rule-based Planning ...............................................................................................36
        2.3.3.5 Theorem Proving .......................................................................................................37
        2.3.3.6 Platforms for Web Service Composition ................................................................39
    2.4 Comparison with Related Work ..................................................................................................40
    2.5 Conclusions ....................................................................................................................................42
Chapter 3.......................................................................................................................................................43
List of Figures

Figure 2-1: OWL-S upper ontology.................................................................13
Figure 2-2: WSMO core elements ..............................................................16
Figure 2-3: WSMX architecture .................................................................18
Figure 2-4: Rules in the Context of the Semantic Web .............................20
Figure 2-5: The framework of the service composition system .................25
Figure 2-6: The solution tree that the ENQUIRER generates for an incomplete-information planning problem ..................................................33
Figure 3-1: Composition rules for sequential composition ..........................46
Figure 3-2: Representing the composite service as black box ......................46
Figure 3-3: The sequential composition of two withdraw Web services ......47
Figure 3-4: The sequential composition of two Web services for a trip organization 49
Figure 3-5: Composition rules for parallel composition ..............................50
Figure 3-6: Parallel composition of online transactional Web services ........52
Figure 3-7: Composition rules for iteration composition ............................53
Figure 3-8: The iterational composition of a withdraw Web service ..........53
Figure 3-9: Composition rules for Repeat-Until composition ....................55
Figure 3-10: A Web service that enrolls students in a specific course ..........55
Figure 3-11: Composition rules for the If-Then-Else composition ..............58
Figure 3-12: The If-Then-Else composition of a Rent_Car service and a Rent_Bike service .......................................................................60
Figure 3-13: Composition rules for the Repeat-While composition ...........61
Figure 3-14: The Repeat-While composition of a Web service for enrolling students in a course ....................................................................62
Figure 4-1: An ontology for a composite service that organizes trips ..........69
Figure 4-2: A plan with correlations between the preconditions and postconditions .......................................................... 72
Figure 4-3: A plan and the insertion of the composition rules .................72
Figure 4-4: A Depth first approach for the generation of a plan for the goal: A ∧ B .......................................................... 77
Figure 4-5: A Breadth first approach for the generation of a plan of the goal A ∧ B .......................................................... 79
Figure 5-1: The control constructs that provided in the OWL-S ontology ..................87
Figure 5-2: The OWL-S ontology for the sequential composition .........................91
Figure 5-3: The graph representation of a Ticket reservation service ....................91
Figure 5-4: The graph representation of a Room reservation service .....................92
Figure 5-5: The graph representation of the sequential composition of a Ticket reservation and a Room reservation service ..................................................93
Figure 5-6: The OWL-S ontology for the Parallel composition .............................93
Figure 5-7: The graph representation of Book shipping service .............................95
Figure 5-8: The graph representation of a CD shipping service .............................95
Figure 5-9: The graph representation of the complex service that produced from the parallel composition of two shipping services .............................................96
Figure 5-10: The OWL-S ontology for the iterational composition .....................97
Figure 5-11: The OWL-S ontology for the Repeat-While composition ..................97
Figure 5-12: The graph representation of a Withdrawal service ............................98
Figure 5-13: The graph representation of the iterational composition of the Withdrawal service ..............................................................................................99
Figure 5-14: The OWL-ontology for the Repeat-Until composition .....................99
Figure 5-15: The graph representation of a service that enrolls students in a course 100
Figure 5-16: The graph representation of the Repeat-Until composition of the enroll service ........................................................................................................101
Figure 5-17: The OWL-S ontology for the If-Then-Else composition ..................101
Figure 5-18: The graph representation of a service that rent cars .........................102
Figure 5-19: The graph representation of a service that rent bikes .......................103
Figure 5-20: The graph representation of the composite service that produced from the If-Then-Else composition of a Rent_bike service and a Rent_car service ....103
Chapter 1

Introduction

1.1 Motivation

In recent years, a large number of Web services have been emerged with the rapid development of the Web. Web services are considered self-contained and self-describing applications that can be invoked over the Web. Until now, many works have dealt with the description, discovery, interaction and composition of Web services. Languages such as WSDL ([CMR’05]) and BPEL4WS ([IBM02]) are used for services description and service composition, respectively. WSDL describes services as a set of endpoints and messages. It is built upon an XML schema for the definition of a vocabulary for a service specification. The endpoints are referred to the operations of the services and the messages define the format of the data to a single request or response in a communication. In the contrary, BPEL4WS is a flow language. In BPEL4WS, the specifications are XML documents and provide information about the port types of the processes, the orchestration of a process definition and the different role that take part in the message exchanges.

Both of these languages specify Web services without defining conditions that must be satisfied over the Web service execution. However, OWL-S ([OWL-S]) provides Web service specifications with the definition of preconditions and postconditions, conditions that must be satisfied in the initial and final state of the Web service, respectively. The preconditions are conditions that must be satisfied by the inputs of the service in order for its execution to begin and the effects are conditions that must hold after the service’s execution and represent the changes that have been made in the state of the world. Nevertheless, none of the above conditions capture assertions that must be satisfied throughout a service’s execution. Assertions ensure the consistent state of the Web service before and after its execution.
The complex and the increasing demands of the users and the inability of the single Web services to achieve user’s demands, result in the development of more useful solutions for composing existing Web services into complex ones satisfying a user’s requirements. A number of approaches have been proposed to solve that problem. Most of the methods that have been proposed are presented the Web service composition problem from the point of view of a workflow problem or an AI planning problem ([RS04]).

For the former, it is obvious that a composite service is similar to a workflow. A composite service is made up of a set of services, basic or composite, that obey a specific control and data flow. In addition, a workflow has to specify the flow of work items. However, dynamic composition requires generating the work plan automatically. The methods that have been proposed in this approach are based on AI planning and theorem proving. These approaches are based on the assumption that each Web service can be fully specified by its preconditions and effects. A service can be regarded as a black box that takes inputs and produces outputs. The inputs and outputs are determined by the preconditions and the effects of the service, respectively. In addition, after a service execution, the world state is changed. The preconditions define the conditions that must be satisfied in the world state before the service execution, while the effects determine the new states that generated after service execution. Given a set of Web services specified by preconditions and effects, the AI planning methods and theorem proving produce a plan, the execution of which is determined the desired goal. The selection of the services for composition is based on inputs / outputs matching and in some cases, in preconditions / postconditions matching. In the methods where the current state is known (Forward approaches), a reasoner for preconditions evaluation is used.

However, in the cases where the current state is not known, methods for reasoning about service composition like precondition-evaluation cannot be applied. In the backward approach, the search is made backward from the goal state, since the current state is not known, the service selection will be based only on the inputs / outputs and preconditions / postconditions matching. Specifically, the matching is based on the types of the conditions. The type matching is not the most appropriate method for the services, which specified by conditional effects. For these services, we cannot determine which one of the conditional effects will finally take place after its
execution. Moreover, reasoning about action interactions that participate in the composition cannot be performed.

1.2 Contributions

In this thesis, we propose a methodology for augmenting the Web service specifications with a set of properties, known as invariants. Invariants must be satisfied in the initial and final state of the service and ensure the consistent state of the Web services. They are assertions over general variables of the service, which must be satisfied both before and after service execution.

In addition, we propose a set of composition rules which determine what must be satisfied among the services that participate in a specific type of composition. The rules can determine the effects of a complex operation such as sequence, parallel, iteration, repeat-until, if-then-else and repeat-while. Considering, a complex service as a black box and the specifications of the appropriate services that participate in the composition, if these services satisfy the appropriate rules, we have the ability to know during the composition, the exact effects of the composite service that they will be produced after its execution. The effects and the outputs of the composite service are determined by the type of the composition of the component services.

The rules provide a powerful technique for reasoning about service composition. Their evaluation in a complex iteration ensures the possible and effective composition of a set of services. Moreover, they can be used for reasoning about the interaction among the services that are selected for the plan generation of a planner that uses backward approach. We propose an algorithm that augments planners based on the backward approach with the composition rules. The planner produces a plan consisted of the sequential and parallel composition of available services. The selection of the services is based on the rule outcome of the evaluation. The satisfiability of the rule conditions can be checked with the use of a theorem prover, which makes the planner sound. The insertion of the composition rules in the Web service specification augments them with compositionality, an appropriate characteristic for reasoning about service composition and verification of the complex Web service. Finally, we choose OWL-S, an OWL ontology for Web service
specification, for specifying these rules. The OWL-S ontology provides services with
preconditions and postconditions, conditions which are required in rule definitions.
For the definition of the invariants in the OWL-S, we propose a new property in the
ontology that correlates each process with a set of invariants. In addition, a set of
changes is proposed on the properties which are defined in the class with instances
composite processes. The definition of the rules made with the use of the SWRL FOL
language ([PaS04]). A SWRL FOL is a language for defining rules in OWL. The rules
in the SWRL FOL have the form: “if A and B then C”, which means that whenever
the conditions specified in the body of the rules hold, then the conditions specified in
the head of the rule must also hold. So, with the use of the SWRL FOL, we can define
constraints among classes of the OWL-S ontology.

1.3 Organization of the document

The thesis is organized as follows:

Chapter 2 is a report on the state of the art on the Web services standards.
Firstly, we introduce the standards that have been defined for service description,
discovery, interaction and composition. A set of languages for rule definition in the
domain of semantic Web is also presented. We present a report about the Web service
composition problem. We analyze the characteristics and the basic components of a
service composition system. We analyze the service composition as a workflow and
AI planning problem.

In Chapter 3 the composition rules, that must be evaluated in each type of
complex operation (Sequence, Parallel, Iteration, Repeat-Until, If-Then-Else, and
Repeat-While) are analyzed.

Chapter 4 presents the methodology for the use of the composition rules in
backward planning.

In Chapter 5 we focus on the SWRL and SWRL FOL languages that specify
rules in the domain of OWL. In addition, a set of changes in the OWL-S ontology,
which must be made in order to define the rules in the SWRL FOL, are presented.
Besides, the changes are pointed out the assumptions which must be made in the rules
due to SWRL FOL and OWL-S ontology definition.
Chapter 6 summarizes the work of this thesis.
Chapter 2

Web Service Composition: State of the Art

Web services are applications accessible on the Web by using standard Internet protocols. They are self-contained, self-describing and can be published, located and invoked across the Web. According to the World Wide Web consortium (W3C) a Web service can be described by the following definition: “A Web service is a software system by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols” ([W3C]). In addition, they can be represented as black-boxes with a specific functionality that can be reused without worrying about the details of implementation. As the Web service approach is based on the notion of “service”, the technology which will be addressed will be what a service is and how can be described.

A great deal of research efforts have found on Web services in the recent years. Several platforms and languages for Web services specification, discovery, interaction and composition have been proposed. Service description is based on interfaces and interface definition languages. Once the services have been described, they must be made available in order to be used. Thus, the services are stored in a repository, in which new services can be registered and also users can search for a specific service which fulfils their preferences. The service discovery can be done at design-time by searching the directory and find the most suitable service, or at runtime by using dynamic binding techniques. The binding problem demands the insertion of a set of tools that will enable the interaction among the selected Web services. Some of the languages that have been proposed for the services description are the Universal Description, Discovery, and Integration (UDDI) ([BCE+02]), Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP) ([Box01]). UDDI is used for listing the services that are available, WSDL is used for
the description of the services and SOAP is a protocol for transferring data between services. These languages provide description, discovery and integration in Web services. Also, there is another set of languages that can be used for Web services specification. These languages provide the Web services with the ability to integrate with other services in order to fulfil the requirements of a user. Indeed, many languages have been proposed in an attempt to establish a common way of defining a composite service. The most credible proposal that has been approved by the main players of the Web services, for defining a composite service is the BPEL ([IBM02]), also known as the Business Process Execution Language for Web Service (BPEL4WS). In addition, DAML-S ([ABH+02]) (formally OWL-S) is an ontology for Web service specifications that enable automatic composition. An overview of Web services standards is presented in detail in the following sections.

### 2.1 Web Services Standards

A set of standards that can be used for Web services specification, discovery, interaction and composition are presented below:

**WSFL**

It is an XML language for the description of Web services composition. It is layered on the top of Web services Description Language (WSDL) which is used for the description of the services interface and their protocols bindings. It specifies how different services ought to be invoked. WSFL ([Lay01]) defines the composition of web services as a flow model or a global model. The two models are described by a public interface and an internal composition structure. The flow model presents the combination of the necessary existing Web service providers. It represents the specification of the activities and their properties. Each invocation of a Web service corresponds with a workflow node called a flow activity. In addition, the global model represents the ability to describe the interactions between existing services and to define new web services as the composition of existing ones.
CHAPTER 2. WEB SERVICE COMPOSITION: STATE OF THE ART

**XLANG**

As WSFL, XLANG is an early work for Web service orchestration. XLANG [XLANG01] describes the behaviour of a single Web service. Its notation is similar with the notation of the workflow languages. The XLANG is used for the creation of the business processes and the interaction between service providers. Its specification is suitable for the sequential, parallel and conditional data flow. The XLANG is using the WSDL for the specification of the interfaces of the processes.

**BPEL4WS**

The specification of the service with the Web Service Description Language describes only the syntax of the messages that are exchanged between the services. The order in which the messages will be exchanged, is not defined by WSDL so, a language that specifies a flow data must be used ([KS03]). Business Process Execution language (BPEL4WS) is such a flow language. BPEL4WS is produced as a convergence between the languages WSFL and XLANG. It has become the standard for the Web service composition providing more functionality than that provided by the WSDL, SOAP and UDDI. Web service integration requires more complex functionality than the WSDL, SOAP and UDDI provide. BPEL4WS can specify both the external behaviour (abstract process) and the internal implementation (executable process). The main difference between the abstract process and the executable is that in the former the way that the message parameters and control flow data determined can be left unspecified. The BPEL specifications are XML documents that provide information such as the port types of the processes, the orchestration of a process definition and the different roles that take part in the message exchanges. In accordance with the above, the definition of the processes and the partners that interact with it are described in WSDL abstract services.

**WSDL**

Web Service Description Language is an XML format describing services as a set of endpoints and messages. Also, it describes network services based on messaging layer like SOAP. WSDL is built upon an XML schema for the definition of an XML vocabulary for the Web Service specification. WSDL specifications are often characterized by an abstract and a concrete part. The abstract part is specified by the port type’s definitions. A port type is analogous to an interface. The operations of the
services (endpoints) are grouped into port types which describe the services abstract such as the location where the operation can be invoked. Each operation defines a simple exchange of messages. A message defines the data format to a single request or response in the communication and consists of one or more parts. The parts can be compared with the parameters that a function has. It provides the description of the exchanges between a service and a node, the location and the operations of the service. All of these constructs are defined in XML. With the Web Services description in WSDL, inputs and outputs are specified as strings rather than mere appropriately typed concept. WSDL specifies abstract types using XML, whereas OWL-S uses classes for the definitions of these abstract types. So, WSDL is unable to express the semantics of an OWL class, therefore the WSDL is unable to express integrity constraints for a Web Service such as preconditions and postconditions.

**SOAP**

SOAP (Simple Object Access Protocol) provides the definition of the XML-based information which can be used for exchanging structured and type information between peers. SOAP is a protocol that underlies all interactions among Web services. It provides a way to communicate between applications running on different operating systems, with different technologies and programming languages. The messages sent among Web services for interaction miss semantics and are one-way asynchronous messages.

**UDDI**

UDDI (Universal Description Discovery and Integration) is also known as “yellow pages” for Web services. UDDI supports the description and discovery of the Web services and the interfaces that can be used for service access. With the UDDI interface, business can be dynamically connected to services provided by external business partners. It is layered over SOAP and assumes that requests and responses are UDDI objects sent around as SOAP messages. The information about the services made available in UDDI includes the name, the telephone and other contact information. It also provides information about the category in which business
2.1.1 Specification languages for Semantic Web services

Web services added a new functionality on the Web by enabling the use and combination of functional components over the Web. The insertion of semantics in Web services enables the automatic discovery, composition, execution and invocation of the services. Two major initiatives have been proposed for the insertion of semantic annotations into Web services: OWL-S and WSMO. These two specification languages are presented in the sections below in detail.

2.1.1.1 Semantic Web Service Description with OWL-S

The Semantic Web enables greater access not only to content but also to services on the Web. Web resources description should enable users with automatic discovery, composition and monitoring of them. OWL-S (formerly known as DAML-S) is an ontology that makes this functionality possible. OWL-S is based on the Web markup languages OWL and its predecessor DAML+OIL ([Hor01]). These languages provide an appropriate, Web-compatible representation framework language for declaring and describing services.

OWL-S structuring is based on the functionality that a Web service must provide to an agent. OWL-S does not intend to replace the current standards for Web services specification, but it augments them with new capabilities. OWL-S relies on WSDL for the integration of the services and augments UDDI with the definition of the Service Profile. Specifically, OWL-S equips a user with three essential types of knowledge about a service.

• “What does the service provide?” The answer in this question can be considered as a service advertisement. This information is presented in the profile, so, an instance of a class Service presents a Service Profile.
• “How is it used?” The answer in this question is captured in the ServiceModel class. This class is linked with every instance of the class Service with the use of the property describedby.

• “How does one evoke with it?” The answer to this question gives us information about the transport protocols which are defined in the ServiceGrounding class.

In general, the ServiceProfile provides the information needed for an agent to discover a service while, ServiceModel and ServiceGrounding provide information about how a service can be used by an agent. These three basic classes for Web services specification will be presented below analytically. As it is referred, OWL-S enables automatic discovery, composition and monitoring. The discovery is a process for location of Web services that can provide a particular class of service capabilities. With the OWL-S markup, the information for Web service discovery could be specified as computer-interpretable semantic markup at the service Web sites and a service engine could be used to locate them automatically. Without the use of the OWL-S markup, a user had to select the appropriate services for composition and specify the composition manually. But now, with OWL-S the information necessary for service selection and composition will be encoded in the service Web sites. In addition, software will be manipulating these representations together with the specification of the objectives of the task, to achieve the task automatically. Finally, OWL-S can be used for the monitoring of the execution of a set of services. This task provides the user with the ability to know the state of his request during a service execution.
Service Profile

The Service Profile includes information of what is accomplished by the service, such as limitations on service applicability, requirements that must be satisfied by the requester in order to use the service successfully, and quality of service. An OWL-S profile describes a service with the representation of three basic types of information. First of all, it gives information about the service provider. Specifically, it describes contact information that refers to the entity that provides the service. Secondly, it provides information about the services functional description. It describes the inputs required by the service and the outputs produced by its execution. The functional description refers to the conditions under which, the service execution can be considered successful and the effects that resulted from its execution.

Finally, besides the functional properties of a service the non-functional ones, which refer to service features such as the category and the quality of the service, are defined. The Service Profile plays the most important role in service selection, but once the service has been selected the profile is useless. The agent then has to use the Service Model in order to control the interaction with the service. The Service Profile and the Service Model have two different roles in service transaction and are considered to be two different representations about the same service. Although the inputs, outputs, preconditions and effects of a service are described not only in the profile but also in the model, there is no constraint about the consistency of these two representations. In other words, a profile can present a service description that can be inconsistent with the model description without producing any problem in the validity of the OWL expression.
Service Model

After the selection of the appropriate service by using the Service Profile, the Service Model is used for the interaction with that specific service. A service can be viewed as a process. A specification of the ways that a client may interact with a service is considered as a process. There are three kinds of processes, atomic, simple and composite. An atomic process is a description of a service that expects one message and returns one message. Atomic processes have no sub-processes and are executed in a single step. A simple process provides an abstraction mechanism to provide multiple views of the same process. Moreover a composite process can be decomposed into other composite or non-composite processes. The decomposition is specified with the use of control construct, such as Sequence, Split, Split+Join, Choice, Any-Order, Condition, If-Then-Else, Iterate, Repeat-While, and Repeat-Until. Each composite process has a property composedOf which refers to a specific control construct. Each control construct is linked with a property components to indicate the nested control construct from which it can be consisted.

A process can have any number of inputs representing the information requested from the user. Also, it can have any number of outputs, information that the service produces after its execution. A process can be described by a set of preconditions that must be satisfied for the successful service invocation. After a successful execution of a process a set of effects change the world state. The outputs and the effects can depend on conditions that must be true of the world state in the time that the process is executed.

Inputs and outputs are subclasses of a general class called Parameter. Preconditions and effects are presented as logical formulas. These logical formulas can be presented with the use of one of the following languages: SWRL ([BDG’04]), DRS ([McD04]), and KIF ([dpA98]). The Semantic Web Rule Language (SWRL) is an extension of OWL, which adds Horn rules over OWL DL ontologies. The Knowledge Interchange Format (KIF) is a standard for the interchange of knowledge between bases. The KIF is an extension of first-order logic with normal text syntax. In addition DRS (Declarative RDF System) provides a vocabulary for writing arbitrary formulas, without defining their semantics.

University of Crete, Computer Science Department
If a process has a precondition, then the process cannot be performed successfully unless the precondition is true. The successful performance of a process results in the change of the state of the world. In OWL-S the effects and the outputs are not linked directly with the class process. This is made, in order to present the dependence among the context, the effects and the outputs. In process, each effect is linked with a condition, under which this effect and a specific output occur. We have to say that there is a fundamental difference between effects and outputs. Effects describe conditions in the world, while outputs describe information produced after the service execution.

Service Grounding

The Service Grounding specifies information of how to access a service. We can say that the grounding defines explicitly the abstract description of those elements that are required for the service interaction such as inputs and outputs. The main function of the grounding is to show how the abstract description of the inputs and outputs of a service are presented as messages which carry those specific inputs and outputs in a specific format. Grounding gives details about the protocols, the message formats, the serialization and the addressing. For message specification, the Web Services Description Language (WSDL) is used.

2.1.1.2 Web Service Modelling Ontology

WSMO ([BF02]), ([KLR04]), ([JFL+05]) is ontology and a conceptual model for the specification of Semantic Web services. WSMO tries to create ontology for Web services descriptions in order to solve specific problems, focusing on the integration problem. It introduces mediators for linking heterogeneous components for modelling Web services. The concepts are Ontologies, Goals, Web services and Mediators.
WSMO does not provide an upper ontology for services description. It describes the goal separately from the Web services and the ontologies. It presents that, as goal and Web service capability. A goal specifies the objectives that a user may have when he uses a specific service. A Web service capability defines the service by means of what the service offers to the requester.

In WSMO, the data transformation is defined with the use of pre-condition and post-conditions, specific conditions over the inputs and outputs of a service. Post-conditions define a relation between inputs and outputs; what a service returns in response to its inputs. In addition, the change of the state is specified by assumptions and effects. The assumptions refer to the state of the world beyond the actual input and the effects to the state after services execution.

The Web service specification plays the role of the process model. It specifies the functionality of the service, the non-functional properties and other aspects. In WSMO, a service can have zero or one capability and zero or multiple interfaces. A Web service interface description is composed of choreography and orchestration. Choreography describes how one web service interacts with another service and orchestration specifies how a complex web service calls subordinate web services or goals. Specifically, it specifies the sequence and the condition in which a Web service invokes other Web services for achieving a function.

As the interface of a Web service can be linked with the orchestration of the sub-processes, the WSMO provides not only static composition but also dynamic. So, composite processes in WSMO can be modelled by defining the orchestration and the proxies of sub services in the interface of the Web service. Simple processes are not
explicitly presented in WSMO, but can be introduced as a proxy that is linked to a Web service with no grounding. Finally, atomic processes can be defined by the capability of a Web service. In addition, a Web service in WSMO also includes grounding. In WSMO, groundings are linked to the Web services with the use of the groundings properties and a Web service can have more than one grounding, but any grounding can refer to only one service.

WSMO follows a layered approach due to the complexity of the domains that the Semantic Web services used. There are three WSMO species: WSMO-Lite, WSMO-Standard and WSMO-Full. WSMO-Lite presents a minimal set of WSMO-Standard. In addition, WSMO-Full tries to extend WSMO-Standard and to incorporate a B2B perspective.

For the definition of the logical expressions that are defined in WSMO goals, mediators, ontologies and Web services the F-logic language is proposed, however no representation language for the definition of the ontology itself is proposed. A logical formalism must be needed for the definition of these conditions. WSML ([dBFL'04]) is a family of languages for the specification of Ontologies and Web services, based on the WSMO conceptual model. WSML presents different degrees of specification. WSML-Core has the least expressive power of all the languages of the WSML. The WSML-Core is defined by the intersection of Description Logic and Horn Logic. WSML-DL extends the WSML-Core capturing full the Description Logic \( SHOIN(D) \). WSML-Flight extends WSML-Core with several features from OWL full, such as meta-classes, constraints and several other features. WSML-Rule supports WSML-Core with Horn Logic. Finally, WSML-Full is the closest variant of the WSMO conceptual model.

Web Services Modelling Execution (WSMX) ([Ore04]) builds on WSMO. It is an execution environment for dynamic discovery, selection, mediation, and invocation of Web services. WSMX manages a repository of Web services, ontologies and mediators. It can achieve a user’s goal by dynamically selecting a matching Web service, mediating the data that needs to be communicated to this service and invoking it. WSMX consists of different components that communicate with events. The communication between the components is asynchronous and that’s because a component can raise an event with parameters and some other component some when it consumes this event and reacts upon it. WSMX has a main component that invokes all the different components in order to achieve the required functionality. WSMX.

Vassiliki Alevizou
accept a request from another application and the manager component tries to achieve it using the discovery, selector, mediator and invoker components. Mediators are used to solve different interoperability problems, like differences in ontologies used by a Web service and a goal. The WSMX architecture is presented below.

![WSMX Architecture](image)

**Figure 2-3: WSMX architecture**

### 2.1.1.3 Comparing OWL-S and WSMO

From the presentation of the OWL-S and WSMO made in the previous sections, it is clear that the WSMO except from the description of Web services it also focuses on the integration problem, thus it inserts a set of mediators that can be used for the integration of the heterogeneous information. The requester and the service needs are viewed in a different way in the WSMO from OWL-S. In WSMO, they are defined separately, while in the OWL-S, the OWL-S profile describes these two concepts together. The OWL-S service profile can be produced with the conjunction of the WSMO goal, the WSMO Web service capability and the Web service non-functional properties.

The inconsistency problem present in OWL-S due to the IOPEs (inputs, outputs, preconditions and effects are normally referred to as IOPEs) which is defined not only in the Service Profile but also in the Process Model is also presented in WSMO. In OWL-S, the IOEPs definition in these two models does not follow a strict relation between them. The only assumption that is made is that the IOPEs of the service profile are subset of the ones defined in the service model. However, in WSMO the wgMediator which links the goals with the Web service capabilities does
not contain any restriction about the consistency of post-conditions and effects that
defined in the goal and also in Web service specification.

While WSMO defines five different variants of the WSML language for the
specification of the logical expressions, the OWL-S does not provide the definition of
the variables over the predicates that determine the relationship between the inputs
and the outputs. However, it provides a set of languages that can be used for
preconditions and postconditions definitions. Specifically, it recommends the use of
SWRL, KIF and DRS. These languages in contrast to WSML language, do not have a
strict relationship with the OWL-S as the WSML with the WSMO ontology. For
example, in the KIF language the expressions are specified as strings, a type that is
not related with the definition of the OWL.

In WSMO, due to its recent proposal and implementation, many of its
features have not been completed yet. For example the WSMO does not yet define
control construct and data flow for composite services, a feature that is important for
the definition and specification of the complex services. Also, the cardinality
constraints in WSMO standard do not allow the definition of more than one value for
the non-functional properties of the service, which might become a big restriction for
some applications.

In conclusion, in accordance to the representation of these two Web services
specification language, the WSMO is more suitable for Web services specification
than OWL-S. But, the incompleteness of some of the WSMO features so far, make it
not suitable for its use for the Web service description.

2.2 Defining Rules in the domain of the Semantic Web

Rules in the Web have become a mainstream topic. They play an important role in
knowledge based systems and in Intelligent Agents. In addition, they are usable for
the declarative specification of Web services. With the use of rules is made possibly
the definition of constraints in the domain of Semantic Web. Many proposals of such
languages have been presented. The Rule Markup language (RuleML) ([RuleML]),
the Declarative RDF system (DRS), and the Knowledge Interchange Format (KIF) are
languages that can be used for rule specification within the domain of the Semantic
Web. In addition the Semantic Web Rule Language (SWRL) and the Semantic Web Rule language First Order Logic (SWRL FOL) ([PS04]) define rules and constraints in the framework of the OWL. Each of these rule languages, with the specification of the rules and constraints, augments a specific domain in the Semantic Web achieving solutions for specific problems. For example the languages SWRL and SWRL FOL provide the OWL-S ontology with the definition of relationships between two properties.

Figure 2-4: Rules in the Context of the Semantic Web

**RuleML**

The Rule Markup Language is an XML-based markup language that permitting both forward and backward chaining for deduction, rewriting and further inferential transformational tasks. RuleML based on the proposed XML languages BRML (Business Rules Markup Language), RFML (Relational Functional Markup Language) and on the XRML (Extensible Rule Markup Language). RuleML uses distinct, standard XML tags to define rule base, composed of facts and rules. RuleML is a language that does not depend on the inference engine and that because rule defining with it must be translated into another inference engine language like Jess (Java Expert System Shell), LISP or Prolog in order to be executed. The design of RuleML is based on the independence of the language from the engines, and the transaction of the data between different engines.
**DRS**

The Declarative RDF system is an OWL ontology that used for the specification of arbitrary formulas as an object. The DRS describes the formulas without presenting the semantics of them. So, when the DRS is used for the specification of a Web service, it must be made an assumption about the semantics of the service. Between the DRS and other rule specification languages such as SWRL and RuleML there are many differences in the philosophy. The SWRL and RuleML define rules, while the DRS is used for the definition of propositions.

**KIF**

The Knowledge Interchange Format (KIF) is a language for the interchange of knowledge among disparate programs. The expressions defined by this language are understandable without the need of an interpreter. It is suitable for nonmonotonic reasoning rules and provides the definition of objects, functions and relations. So, the KIF can be used for the interchange of knowledge among heterogeneous systems, also for representation of the knowledge in ontologies and knowledge bases and for the specification of the expressions that can be used as input and outputs of inference engines. KIF is consisted from three parts:

- Part 1 (First-Order): specifies the semantics and the syntax of a First-Order language with equality.
- Part 2 (Infinitary KIF): is an expansion of the First-Order KIF.
- Part 3 (MetaKIF): is an expansion of the First-Order KIF-Core.

**SWRL**

SWRL provides a high-level abstract syntax that extends the OWL abstract syntax. It is based on the combination of the OWL DL and OWL Lite sublanguages of the OWL language. SWRL extends OWL DL with Horn rule, so the OWL DL abstract syntax augmenting with a further axiom:

\[
\text{axiom ::= rule}
\]

A rule axiom consists of an antecedent (body) and a consequent (head) each of which consists of a set of atoms.

\[
\text{rule ::= 'Implies(' [ URIreference ] \{ annotation \} antecedent consequent \')'}
\]

\[
\text{antecedent ::= 'Antecedent(' \{ atom \} \')'}
\]

\[
\text{consequent ::= 'Consequent(' \{ atom \} \')'}
\]
Atoms can be of the form $C(x)$, $P(x,y)$, sameAs($x,y$) or differentFrom($x,y$). The $C$ is an OWL description, $P$ a property and $x, y$ variables. An atom may refer to individuals, individual variables, and data variables or data literals. Only variables which are defined in the antecedent of a rule can occur in the consequent, so the scope of the variable is limited at the border of the rule in which is specified.

\[
\text{i-object ::= i-variable | individualID}
\]

\[
\text{d-object ::= d-variable | dataLiteral}
\]

A rule in the SWRL has the following form: “\textit{if } A \textit{ and } B \textit{ then } C \textit{”}. The meaning of this rule can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. When an antecedent is empty, is treated as trivially true, but empty consequent is treated as trivially false.

\section*{SWRL FOL}

Such as SWRL, SWRL FOL can be used for rules definition in the properties of the OWL-S ontology. SWRL FOL uses First-Order-Logic RuleML for the definition of these rules. SWRL FOL subset of SWRL FOL RuleML comprises an extension of the SWRL. It extends the set of logic OWL axioms with an axiom for arbitrary first-order formula with a unary or binary predicates.

The abstract syntax of the SWRL contains a sequence of axioms and facts. This abstract syntax is extended with assertions axioms that contain first-order logic.

\[
\text{assertion ::= 'Assertion(' [ URIreference ] {annotation} foformula {foformula} ')'}
\]

The formulae in assertion is based on the first-order formulae allowing the conjunctions and disjunctions to have any number of conjuncts or disjuncts.
The types of the variables in the “Forall” and “Exists” constructs are specified from the following formulae.

\[ \text{variable ::= } \text{I-variable(' URIreference description ') | D-variable(' URIreference dataRange ')} \]

As in the SWRL, the rules in the SWRL FOL have the same form: “if A and B then C”, and the same meaning.

### 2.3 Web Service Composition

A Web service which is provided by the combination of the functionality of other services is called a composite service. The process followed for the implementation of the composite service is known as service composition. Although, many studies of the composition of services have been preformed, it still remains a huge problem. A Web service composition model can be specified by the following six dimensions: component model, orchestration, data and data access model, service selection, transactional model and exception handling ([ACK'04]). The component model makes assumptions about the components that participate in the composition. The orchestration model defines the order that the process will be invoked. Specifically, it determines the control flow and the data flow among services. Many orchestration models have been proposed for the specification of the order of the services invocation such as UML activity diagram, statecharts, activity hierarchies, Petri-nets, \( \pi \)-calculus. In addition, in the composite process the data and the way that they are accessed must also be defined. For example, the data can be defined as an XML message that can be passed among services. The data access determines the way that the data is passed from one Web service to another, specifically, among the services

Vassiliki Alevizou
that are being composed. In order for the data to be passed from one Web service to
another must be known the specific service that is the target of the message. The
binding of the service can be static by defining the URL of the appropriate services or
can be made dynamic. Finally, the exception handling defines what must be made in
case where an undesirable situation has been occurred. That can be resulted from the
wrong execution of a service or from a failure of a system. With the exception
handling, this situation is handled without resulting in the termination of the
composite process.

The ability to perform automated or semi-automated composition is an
essential step to decrease time and cost in the development and maintenance of
complex services. Most of the techniques that have been proposed for service
composition, deal with low-level programming details. Thus users must know details
about how processes are interconnected, services are invoked and how messages map
to one another.

The automatic composition provides automatic selection and interoperation of
Web services. The semantics of Web services would be crucial to enable Web service
composition. With this kind of composition the user would have to specify only
“what” the composite service has to do without being concerned about “how”. “What”
is determined by the features of the composite service. The features may be syntactic
(e.g. the number of the parameters) or semantic (e.g. the functionality offered by the
service). While researchers have devoted a lot of efforts to solve the automatic
composition problem, it still remains an open problem. To perform service
composition one must be able to select the most appropriate services for composition
but also use the new services that produced at runtime, considering the new
information the system will have. There have been many methods proposed in order
to solve the web services composition problem considering the above difficulties.
The problem of the Web service composition can be seen either as a workflow
problem or as a planning problem.
2.3.1 Web Services Composition Framework

In this section, we will present an abstract framework that can be used for web services composition. In this framework there are two kinds of participants, service providers and service requesters. The first advertise Web services for use and the second consume information or services. The framework consists from the following basic parts:

![Figure 2-5: The framework of the service composition system](image)

**Translator**

The translator translates the standard Web service languages, such as WSDL and DAML-S, in an internal language that can be used by the process generator in order to produce the composite process, which indulges the requirements of the user. The external languages that are used for the service specification provide accessibility to the user in what they want and what they offer. However, the process generator requires more formal languages such as the logic programming languages in order to produce the complex service.

**Process generator**

The process generator selects the appropriate services provided in the Service Repository in order to indulge the requirements of the requester. The requester specifies his goal in a service specification language. The generator selects the services in accordance to their functional properties such as inputs, and outputs. The methods, which can be used, for the service composition are based on the proposed
planners that are used for the generation of a plan according to the goal, the initial state and the users preferences. If there are more than one plans, these are evaluated from the evaluator and the best is proposed.

**Evaluator**

The process generator can produce more than one composite service that fulfils the requirements of the user. Thus, all these composite services are evaluated by the non-functional properties. The non-functional properties are referring to the quality, the category, e.t.c. The user has to specify weights into non-functional attributes, so the most appropriate service is selected.

**Execution engine**

After the selection of the composite service, the execution engine executes the service. The execution is determined by passing a sequence of message among the services that participate in the composition.

**Service repository**

The service provider advertises its services. There are several languages that can be used for service advertisement, such as UDDI and OWL-S Service Profile. These languages provide information about the functional and non-functional properties of the services. The functional properties are determined by the inputs, outputs, preconditions, and postconditions. They determine what a service expects from a user with the definition of the inputs and the preconditions. Also, they determine what they provide to the user after their execution such as effects and outputs. The non-functional properties such as service category and quality determine general characteristics of the service.
2.3.2 Workflow problem

We can say that a service composition is similar to a workflow in many ways. A composite service is specified by a set of services, a control construct and a data flow that illustrates the messages which changed between the services. Also, a workflow specifies the flow of work items. The workflow composition methods are used when the process model is given and the selection of services is required, the execution of which indulge the user requirements. These nodes will be regarded as abstract services that must be replaced with the concrete services at runtime. On the other hand a planner is used when only the requirements and the constraints of the user are known.

The eFlow ([CJ00]) is a system that provides the specification, enactment and management of composite e-services. It contains a rich set of features, such as events and exception handling, ACID service-level transactions, security management and others. The eFlow the composite service is modelled as a process. Each composite service is modelled as a graph that presents the order of the execution among the nodes of the process. The graph can contain three types of nodes, service, decision and event nodes. The service nodes represent the invocation of basic or composite services. The decision nodes determine the execution flow by defining alternatives and rules. Finally, the event nodes enable service processes to send and receive different types of events. The arcs of the graph contain label with predicates, the value of which is determined by the process data after executing the specific node. If the value of this predicate is true then the nodes that are connected with these outgoing arcs are executed. Due to frequent changes in the environment, e-flow provides several features for achieving this goal. Each service node contains the description of the service which must be invoked by defining a service selection rule.

Also, eFlow provides the dynamic service node creation. This is achieved by including the notion of generic service node in its model. The generic service nodes are not bound in a static set of services. In addition, generic nodes contain a parameter whose value is a set of actual service nodes. Parameter can be set at the instantiation time or at runtime. Finally, it provides dynamic service process modifications. The users, specifically the authorized users, have the ability to make modifications in the instances of a service process. The modifications can be referred to either a specific
instance of a service process or to a set of instances of the same service process that has the same properties. Specifically, the user can modify every aspect of the schema, including the flow structure, the definition of service, decision and event nodes, process data and even transactional regions.

### 2.3.3 Planning problem

One family of techniques that has been proposed for the dynamic composition of services is the AI planning. Planning is a search problem that requires finding an efficient sequence of actions that transform a system from a given starting state to the goal state. By ([NR95]) the problem of planning is characterized as follows: “Planning can be interpreted as a kind of problem solving, where an agent uses its beliefs about available actions and their consequences, in order to identify a solution over an abstract set of possible plans”. Generally, a planning problem has the following components:

- A description of an initial state
- A description of the possible actions which may be executed
- A description of a desired goal

A planning problem can be described as a five-tuple \(<S, s_0, G, A, \Gamma>\) ([CST03]). \(S\) is the set of all possible states of the world. \(S_0\) denotes the initial state of the planner, which belong to the set \(S\). \(G \subseteq S\) denotes the set of the goals that the planner should indulge. The variable \(A\) denotes the actions that can be used from the planner in order to reach a goal state. Finally the symbol \(\Gamma \subseteq S \times A \times S\) denotes the semantics of each action by describing a state that is produced when a specific action is executed in a particular world state.

By seeing the plan from the classical point of view, the planner gives as a solution a sequence of actions, the execution of which leads to a state that satisfies the desired goal. It is sufficient that these classical plans do not capture complex problems such as nondeterminism. There exist several planners that produce conditional plans in order to capture complex goals and domains. We can say that these planners encounter effect from the observers of the environment. These planners produce a
plan that contains all the possible contingencies that could arise. So, at runtime the agent selects the appropriate branch from this conditional plan depending on its situation. The conditional planners are also known as contingency planners. A conditional planner is appropriate when uncertainties in the environment preclude the selection of a single course of action to accomplish a goal ([PS92]). These kinds of planners do not use the method of replanning at runtime, but they develop projected contingency, one of which is selected at runtime.

As it is said above, a plan consists of an ordered set of actions known also as a step. A plan can be totally ordered. This means that every step is ordered with respect to other steps. In the case of a partially ordered plan, the steps can be unordered with respect to each other. One can say, that a partial order plan is more preferable than a totally order due to the flexible execution of the first. However, a flexible execution is determined not only by the type of the plan but also by the search strategy and the search heuristics implemented.

Depending on the starting point of the search in order to generate a plan we distinguish the forward chaining planner and the backward chaining planner. The Forward Chaining algorithm, known also as progression, begins to search from the initial state. In each state, it tries to find the set of all applicable operators for the current state. For every state that is reached, only the shortest path is recorder from the initial state to this state. This algorithm stops if the goal state is reached. The Backward Chaining algorithm, known also as regression, searches backward from the goal state. As the forward chaining algorithm, this finds all the set of all the applicable operators that can reach the current state. If the initial state is reached, the algorithm terminates. The plan which is produced from this algorithm is a partial-ordered plan in which the actions are partially ordered.

There are several categories of planners that can be used for the generation of an appropriate plan in order to solve a specific problem. These categories will be presented in the following sections.
2.3.3.1 Planning Domain Definition Language (PDDL)

The Planning Domain Definition Language (PDDL) ([GHK'98]) is a language for defining domains and problems for input to automated planners. It has become an acceptance standard for classical planners. PDDL is used for the description of the “physics” of a domain. The main syntactic features of the language are:

- Basic STRIPS-style actions
- Conditional effects
- Universal quantification over dynamic universes
- Domain axioms over stratified theories
- Specification of safety constraints
- Specification of hierarchical actions composed of subactions and subgoals
- Management of multiple problems in multiple domains using differing subsets of language features

The PDDL cannot be considered as the most useful language for planning domain specification and because it provides information about the “physics” of the domains only, such as what the predicates are, what the actions are about and what effects they have and also about the structure of the compound actions. Specifically, it does not provide any information about the cases in which the actions can be used, when a compound action can be selected and under what circumstances. For that reason PDDL has been factored into subsets of features known as requirements. The requirements of a domain defined in PDDL determine if this domain can be used by a specific planner. For example, a domain, which requires conditional effects, cannot be used by a straight STRIPS-representation planner. PDDL can be considered as the de facto language for the classical planning. PDDL

In ([Hal03]) is presented how a temporal planning can be made with a Non-Temporal Planner. Temporal planning is the same with classical planning but in the first the actions are not momentary, they have duration. So, in these types of planners the mode of the time is inserted. An important role in the temporal planning is played by concurrence, where two or more actions can be executed simultaneously. Temporal planning can be described as the merging of classical planning and scheduling. In this work, a system that takes as input a domain and a problem file is presented. This information is presented with the use of the PDDL2.1 ([FL03]) and extension of the
planning domain definition language (PDDL). Initially the system converts the
domain file into STRIPS like domain since the temporal aspects have been removed
from this file. After the conversion the domain is inserted in a classical planner which
produces a total order plan. In this plan it is checked if similar actions can be executed
simultaneously converted into a partial order plan. This partial order plan with the file
that contains the duration aspects is inserted into a program which estimates the
relative and actual timing and produces a valid temporal plan.

2.3.3.2 Hierarchical Task Network Planning

HTN ([EHN94]) is an AI planning system that creates plan by task decomposition.
The intention of each planner is to decompose the initial task into primitive tasks that
can be executed directly. This planner assumes a set of operators that achieve certain
effects when its preconditions hold. These operators are also called primitive tasks.
Besides the primitive tasks, the planner supports a set of methods that presents how a
complex task can be decomposed into some set of subtasks. The HTN planner
produces a sequence of primitive tasks that perform the desirable task. This task can
be a set of partially order tasks that must be achieved.

A variant of HTN planning is SHOP2 (Simple Hierarchical Ordered Planner).
SHOP2 ([HNP’03]) describes a method for translating process models for each Web
service into SHOP2 operators and methods. SHOP2 is a domain-independent HTN
planning system. The difference between the SHOP2 and the other HTN planners is
that SHOP2 generates the steps of the plan in the same order that those steps will be
later executed, and that is because the goal of the planner may be a set of composite
tasks. The ability of the SHOP2 planner to produce a sequence of operators in the
same order that these will be later executed makes it possible to know the current state
of the world at each step of the planning process. Knowing the current state makes the
SHOP2 able to incorporate with significant reasoning mechanisms for precondition-
evaluation ([HNP’04]). SHOP2 planner uses a prover for precondition-evaluation, in
order to determine if a specific operator can be used in this current state. According to
the authors of ([HNP’03]) “A planning problem for SHOP2 is a triple (S, T, D),
where S is the initial state, T is a task list, and D is a domain description”. By taking

Vassiliki Alevizou
(S, T, D) as input, SHOP2 will return a plan $P = (p_1p_2…p_n)$, a sequence of instantiated operators that will achieve T from S in D. The domain D is produced from the translation of the process models for each service into operators and methods with the use of the relevant algorithm. The services that participate in this planning problem must follow a set of assumptions. The Web services with outputs and effects must be divided into a set of services that either provide information (information-providing services) or change the world (world-altering). This means that the services that can be used can have either outputs or effects. In addition the SHOP2 planning system cannot handle concurrency problems, so the composite process cannot contain Split and Split+Join control constructs. And finally the effects of the processes cannot be conditional.

Knowing the domain D, we can translate a DAML-S Web service composition problem into a SHOP2 planning problem. In this approach the assumption that the information that it got during the planning will not be changed is made. This is not always true, because the information can be changed during the planning, so the planner can take into account all the possible changes that have been made into the information. As it is referred above atomic processes can have either outputs or effects, so the information-providing services are defined explicitly in this planner. These services due to the assumption that the information is not completed can be interleaved during the planning. The SHOP2 planer produces a sequence of world altering methods.

SHOP2, with the interleaving of the information-providing services during the planning, augments the initial state of the world with more information that will be useful in the execution of others Web services.

The planning system SHOP2 waits the answers of the information gathering services in order to continue the generation of the appropriate plan. However the execution of the information-providing services sometimes may take longer than the time that will be spent by a planner to generate a plan. Also, there is a possibility of not completing the service execution. However, it is not beneficial for a planner to wait until the appropriate information gathering service terminates. It must continue to look for alternatives plans that do not depend on the information which will be returned from the gathering services.
This problem is faced with the proposition of an HTN-planning algorithm (ENQUIRER) ([HKN’04]) that was designed for planning domain in which the information about the initial state of the world may not be complete. This algorithm extends the way that the planning system SHOP2 deals with the information gathering during planning process. ENQUIRER starts with an incomplete initial world state, executes information providing services during planning and continues to generate alternatives plans while waiting for information to come in. The algorithm takes an incomplete information planning problem \((J, A, T, D)\) as input, where \(J\) is an incomplete state, \(A\) is an askable list, \(T\) is a task network and \(D\) is an HTN-domain description.

“An incomplete-information planning problem \(P^I\) is consistent with a complete-information planning problem \(P^C\) if only if \(S\) is identical with \(J \cup \delta(A)\)” ([HKN’04]).

The solution of this algorithm is a tree composed of all the alternatives plans, which can be generated for the composite task \(T\). This tree is presented in the following figure. The “OR Branch” indicates the alternatives methods that can be used for the precedent task execution, while the “AND Branch” indicates task decomposition.

![Figure 2-6: The solution tree that the ENQUIRER generates for an incomplete-information planning problem](image)

In ([desFS03]) another planner that composes services described in DAML-S into a composite Web service is presented. This planner is a simple backward-
chaining algorithm that generates a plan considering the goal and a set of basic services. As Web services described in DAML-S, the Service models contain the inputs, outputs, preconditions and effects of services. The first step of the planner is to convert all the Service models into Verb-Subject-Object (VSO) triples. All these triples are asserted into the JESS KB ([FrH02]) as facts. After that, for each service is building a set of planning operators. These operators are produced with the help of a set of rules and queries that transform the corresponding triples into a set of facts that form the planning operator. Knowing the operators for each basic service, the planner tries to find the services that satisfy any of the existing goals and add them to the plan. The inputs and preconditions of each operation added to the plan are converted into a new set of goals. A plan will be found, if the outstanding goals are external inputs and preconditions. As external inputs and preconditions, the inputs and preconditions that are provided directly by the agent are known and are not created as outputs and effects of other operators.

We can say that this approach imposes a partial order in services execution. It is assumed that only one service satisfies the appropriate goal each time. If there is more than one service which can be used for the goal satisfaction, it will be suitable to apply heuristics methods for selecting the appropriate plan.

In ([HPS02]) a prototype is presented which can be used for the semi-automatic composition of Web services. This prototype consists of two components, a composer and an inference engine. A composer gives the opportunity to a user to produce a workflow of services. Specifically, the composer is the user interface that handles the communication between the human operator and the engine. The composer presents all the available services in the system and filters the results based on the constraints which the user may specify on the attributes of the service. The matching and the selection of the services are done using the functional and non-functional properties of the services. These properties are defined in the Service Profile used in order to determine if two parameters are matching or not. There are two types of matches between services, a generic matching and an exact matching. A generic match is made when an output type of a service is a subclass of the input type of another service. However, the exact match is made when two parameters are restricted to the same OWL class. As it is obvious the most preferable kind of match is the exact. The matching services are gathered in a list, which contains the top the
services described for their exact match. The composer presents only the services whose outputs could be fed to a selected service as an input. The non-functional properties are used for the selection of a service from the list with all the possible matches. As non-functional properties, we consider the name, the location, the cost and a set of other properties. The inference engine is an OWL reasoner built in Prolog and it stores the information about know services in a Knowledge base (KB).

The composition which is made from this prototype can be described as a DAML-S Composite Process, which can be used for advertising, discovery and for composition with other services.

### 2.3.3.3 High-level Program Execution

In the high-level program execution approach, the idea is to find a sequence of actions that constitutes a legal execution of a given high level program.

Golog ([LLL94]) is such a high-level logic programming language based on the situation calculus. It is suitable for the specification and execution of complex actions in dynamic domains. This approach is alternative to planning and does not change the computational complexity of the task of generating a composition. Moreover, it reduces the search space making the computational advantageous. Golog builds on top of the situation calculus and provides a set of logical constructs. The abbreviation Do(\(\delta, s, s'\)), says that is possible to reach situation \(s'\) from situation \(s\) by executing a sequence of actions as specified by \(\delta\), which is a complex action expression ([MS02]). The language constructs that provides the Golog are: primitive actions, test actions, sequence, nondeterministic choice of two actions, nondeterministic choice of action arguments and nondeterministic iteration.

Given a set of situation calculus-based domain axiomatization Axioms, an initial situation \(S_0\) and a Golog program \(\delta\), the planner has to find a situation \(s\). Specifically, the planning task is to find a sequence of actions \(a_1, \ldots, a_n\) the execution of which results in the situation \(s\).

\[
\text{Axioms} \vdash (\exists s) \text{Do}(\delta, S_0, s)
\]

An extension of the Golog language can be used for the automation of Web service composition ([MS01]) ([MS02]). Supposing that a set of Web services and the
task or the goal that we want to achieve, are known we want to find a composition of services that achieves that task. In this approach a reusable and high–level generic procedure is constructed that will be kept in generic procedure ontologies so users can access them. With the word generic is meant that the program will be generic enough to the needs of different users. Thus the programs must have been some nondeterministic to enable a variety of different choice points to incorporate user’s constraints. This vision is made possible with the adaptation and extension of the logic programming language Go log.

In ([MN02]) a transformation of OWL-S to situation calculus is given. The DAML+OIL has a well-defined semantics, but it is not sufficiently expressive to characterize all and only the intended interpretation of DAML-S. A solution to this problem is given by the translation of OWL-S to first order logic (situation calculus). The OWL-S processes can serve as the desired processes and the atomic or complex actions offered by the Web services. Thus, the composition problem is to find an execution of a Go log program that satisfies the properties defined in the goals.

2.3.3.4 Rule-based Planning

Until now, very little techniques check the composability of the services, which will be invoked in a composition. Composability means the process of checking if Web services to be composed can actually interact with each other. Medjahed ([BEM03]) presents a method to generate composite services from high level declarative description based on the composability of the services. In this method the desired composition is specified with the use of the Composite Service Specification Language (CSSL). For the selection of the appropriate services for user specifications, the composability rules are used. These rules rely on the semantic and syntactic properties of Web services. The syntactic rules are based on the binding protocols for services interaction, while the semantic rules are referred to the composability of the inputs and outputs of the services. Also, they relate to their quality considering the requirements of the user and the soundness of the composition. By the definition “Soundness of the composition” we mean if the composition is worthwhile or not.
Specifically, by sound they mean if the composition of two services provides an added value or not. Thus, they introduced the notion of the composition templates. Each composition template is associated with a composite service. In addition, in order to check a composite service for sound, they provide the stored templates. These templates are defined into two kinds of groups. Templates which are predefined by domain experts, and templates which are “learned” by the system. Every time, that a new composite service is produced, its template is added to the repository with the stored templates. After the generation of the plans, if there are more than one, the most appropriate considering the quality of the composition is selected.

SWORD ([PF02]) is a toolset that provides the developer with the ability to compose base services in order to produce a composite service using rule-based plan generation. The services which are used for the composition belong to the category of information-providing and are specified with the use of an Entity-Relation (ER) model and not with an emerging standard such as WSDL or DAML-S. In order to produce a composite service, the developer specifies the inputs and the outputs of the composite with the use of the world model. Each Web service is specified as a Horn rule where the preconditions of it imply the postconditions. The SWORD determines if the composite service can be produced by the existing services and if so, it generates a plan for it. The plan generation is based on a set of rules defined from the data inputs, conditional inputs, data outputs and conditional outputs.

2.3.3.5 Theorem Proving

Another method for automatic composition of semantic Web services is the use of the Linear Logic theorem proving ([KMR03]) ([KMR04]). The main difference of this method from others is that it considers not only the functional attributes of a service but also the non-functional during the planning. As non-functional attributes the quality, the cost and other attributes that specify the service are referred. The idea is that given a set of services and a set of functional and non-functional attributes, the method must find a composition of atomic services that satisfies the requirements of the user. The requirement is presented by the following formula in Linear Logic ([KMR03]):

Vassiliki Alevizou
$\Gamma; A_1 \vdash (I \otimes o O) \otimes A_2$

$\Gamma$ is a set of logical axioms that represents atomic Web services. $A_1$ is a conjunction of non-functional constraints. $A_2$ is a conjunction of non-functional results. I represents the set of input parameters and O the output parameters.

The “I–oO” represents the outputs parameters O that must be produced considering the input parameters I. In [KMR04] the preconditions and postconditions of the Web services are also taken into consideration. So, the above formula takes now the following form:

$\Gamma; A_1 \vdash ((I \otimes P \otimes o O \otimes F) \otimes E) \otimes A_2$

The requirements and constraints of the user are translated into linear logic. Also, the DAML-S descriptions of the services are translated in Linear Logic with the use of a DAML-S Parser. This approach has been applied to the composition of the value-added services. They divide the services into sets, the value-added services and the core services. The value-added services stand as complements to the core services. For example in online shopping, the product search, the payment and shipment refer to core service while services such as measurement exchange refer to value-added services. In this paper, the problem is restricted to composition of value-added services only assuming that the core service already exists. It has been mentioned that for the composition of core services a business model should be provided which will define the relationships among core services. The user requirements are converted into Linear Logic axiom and try to be proven by a Linear Logic prover which uses the Web services specified into Linear Logic. The prover uses inference rules in order to construct the proof.

The ([Wal00]) presents the problem of service composition as a theorem proving problem. Specifically, this approach is based on the automated deduction and program synthesis. Each agent is associated with a set of axioms that describe the services that it provides. In addition the requirements of the user (known also as query) are expressed as a theorem in the same application-domain theory that the agents have been defined. Considering a set of axioms composed by the agents and the user specification, a theorem prover is used for proving the query. In this approach, the SNARK theorem prover is used. The prover will return a proof with the agents which must be used. This proof can be converted in the Web services specification. The way that is used for the generation of the proof is sufficiently
constructive, meaning that in order to prove the existence of something, a method of finding must be indicated. During the proof generation the execution of agents is possible for providing the information needed.

### 2.3.3.6 Platforms for Web Service Composition

In ([AAM’04]) two techniques for semi-automatic Web services composition are presented. These techniques are based on the inputs and outputs description by an ontology. In the Interface-Matching Automatic (IMA) composition approach ([AAZ03]), different services are put together in order to produce the expected user’s outputs. The optimal composition that is found is based on factors of semantic machining between inputs and outputs and quality of services criteria (QoS) defined by the user. The result produced from this technique is a sequence of Web service, the execution of which provides the user’s goals. In this approach the problem of mismatch between the inputs and the outputs of the services is inserted. In this technique all the possible sequence of services which can be used for the production of user’s goals are produced. From the implementation of the algorithm steps a new Web services ontology is produced from the inputs and outputs matching. In this ontology, nodes represent services while edges represent relations between outputs and inputs of the services. Each edge has a weight, which is determined by the execution time, the semantic similarity value and a factor (λ) defined by the user:

\[ W = (\lambda) \times \text{execution time} + (1-\lambda) \times \text{similarity value} \]

The composition algorithm aims to find the optimal collection of services depending on the execution time and the matching of the parameters.

In Human-Assisted composition (HA) technique that is based on iteratively composing the service by making use of filtering and ranking based on user-provided constraints. Specifically, in each stage, the user selects a service from a ranked list. In the IMA, the user is not involved in the composition specification in which predefined templates are not used. However, in the HA composition, the user is involved in the composite specification, which is based on predefined templates. The HA composition approach has the incrementally generation of a composition path in a customized way as a goal. The HA approach tries to insert the following issues: The
selection of the classes and the instances will be made not only by the matching between the inputs and the outputs but also by the attributes. A ranking and filtering of composable services at certain points is used by the user. The HA uses predefined templates, by removing non-relevant services by services that match their inputs and outputs or adding services that have been suggested by the system.

### 2.4 Comparison with Related Work

In ([CZS04]), ([CSZ04]) a framework that enriches semantic service descriptions with two compositional assertions: assumption and commitment is proposed. The Assumption – Commitment (A-C) framework was first introduced by Jayadev Misra and Mani Chandy ([CM81]) as a proof technique. The insertion of these assertions in Web services description enables reasoning about service composition and verification of their integration. An A-C formula has the following form: $\langle A, C \rangle : \{\varphi\} P\{\psi\}$. The P denotes a process while the A, C, $\varphi$, $\psi$ denote predicates.

In this work the A denotes assumptions describing the expected behaviour of the environment of P. The C denotes commitments that are guaranteed by the process P while the environment does not violate the assumptions, while, the $\varphi$ and $\psi$ express preconditions and postconditions, respectively. The assumptions and commitments are temporal properties that must be indulged for the right execution of the process. In case, where the process is passed by a state which violates these properties, the process execution terminates. For each type of composition (sequence, parallel, iterate, repeat-until, if-then-else) a set of conditions that must be indulged by the services that will be composed has been defined. The conditions determine the assertions that must be satisfied in the composite service and the relations between the conditions of the subservices. These rules insert the compositionality in the composition of the services and they determine that following these conditions the composition of specific services will be sound. Specifically, these assertions characterize the ongoing behaviour of the services. The definition of these temporal properties improves the verification of the composed service by checking at runtime the assertions that require to be held by every service in the composition. They are
interested in verification at runtime, when services are actually wired with each other at the ports. Since the assertions are temporal properties of services and their environments, the proof obligations for the specification of the composition have to be validated by an engine for temporal properties. AnaTempura is such a tool.

The properties are specified in the Interval Temporal Logic (ITL) ([ITL]). Finally, it is shown how the temporal properties can be inserted in the OWL ontologies with the use of the SWRL language. In the contrary, the composition rules that we propose are determined by the conditions of the service, and they ensure that the composite process which will produce following them, will always give as the desired goal without need to execute the services.

In addition, in most works the reasoning about service composition is based on the matching between the inputs, outputs and preconditions, postconditions. Only in the cases of a partial order of a hierarchical task network planning ([HNP'04]) are taken into consideration the action interactions. Specifically, only in the cases where the current state it is known is able to apply reasoning methods for services selection. So, the planners that use constructive methods for plan generation can use reasoning methods. In ([desFS03]) where is used the backward approach for the plan generation, the selection of the services based on the matching between inputs / outputs and preconditions / postconditions. Specifically, they made the assumption that the postconditions that will be produced after the service execution will be always the desirable, ignoring the alternatives. The algorithm terminates, when the inputs and the preconditions of the services are external.

The works that based on theorem proving methods use reasoning for service selection and these planners can be characterize as sound and complete. Specifically, in ([KMR03]) ([KMR04]) the construction of a plan is based on a theorem prover for Linear Logic formulas. These works can be used for the composition of value-added services with a core service that has been already selected by the user. In order these works to be used for the composition of core service must be provided with a business model, which specifies the correlation among the core services. Also, in ([Wal00]) the generation of the proof is sufficiently constructive. This work tries to compose Web pages that have been converted in agents in order to answer a query.
2.5 Conclusions

Concluding, this chapter has aimed to give an overview of the Web service standards that used for the Web service specification, discovery, integration and composition. It is made clear, that even if they have been proposed a lot of works for the automatic web service composition still this problem remains huge.

In the next chapter (Chapter 3), we introduce the composition rules that must be satisfied between the services that participate in a specific type of composition. The rules consisted of predicates that determined by the conditions (preconditions, postconditions, invariants) of the Web service. The rules can be used for reasoning about service composition and interaction action. They can be used for reasoning about service composition at design time without needing the service execution. The evaluation of them is efficient not only in the cases where the current state is known but also in the backward approaches where the generation of a plan is made from the goal state and thus the current state is not possible to be known.
Chapter 3

Definition of Composition Rules

In this work, as it has been referred, we augment the Web service descriptions with of a new type of constraints and also we introduce a set of composition rules that augments the Web services specification with compositionality.

Until now, each Web service is described by a set of preconditions and postconditions, constraints, which refer to the initial and final state of a service, respectively. The preconditions prescribe the beginning of a service execution, while the postconditions define the results that will be produced after its execution. These kinds of constraints refer to conditions that must be satisfied by the inputs and outputs of a service. For example, a Web service with which a user can reserve tickets for a specific flight may have as precondition a constraint that checks the validity of the date that the user insert. Also, it may require from the user to be registered in order to provide him the information. In addition, the postcondition of the service will be the confirmation that the tickets reservation is made or not.

Both of these types of constraints are not correlated with general variables of the Web service, conditions that provide the Web service to be in a consistence state. Specifically, assertions determine the state of the system before and after the execution of the Web service. For example, a global assertion for the service “Ticket Reservation” will be the following condition: “Always, the reservations which have been made until now for a specific flight must be smaller than the possible reservations (total_{booking} < limit_{booking})”. This constraint ensures the consistent state of the service. In other words, it guarantees that the execution of the service is effective to be made, due to the available seats. But also, it ensures that the state of the system after the service execution will not be inconsistent. For example, after the service execution, the seats that have been booked won’t be more than the seats the airplane has. This type of constraints, are called Invariants and are also known as Integrity Constraints in the world of databases. As it is referred, the Invariants determine the consistent state of the system before and after the service execution, thus the initial
and final state of the Web service. An initial state of a service must be consistent in order to be effective with its execution. In addition, the final state produced after the service execution must be consistent, something that results from the right execution of it. It is common knowledge that a Web service can be passed from many inconsistent states until it terminates. In those inconsistent states of the service, the Invariants are not satisfied. This case does not cause us any problem. By representing a Web service as a black box the most important states of it for which we need to have knowledge are the initial and final state. These states determine the initiation of the service execution and the correct termination of it. In both states the Invariants must be satisfied in order to ensure that the system is in a consistent state.

Generally, the efficient execution of a Web service is determined by the proof of the following rule:

\[ Pre \land Inv \land Post \Rightarrow Inv' \]

where Pre, Inv, and Post are the preconditions, invariants and postconditions of a service. The Inv' evaluates the invariants with respect to the state in which postconditions (Post) are satisfied. Thus, each invariant is satisfied by a single Web service. This rule determines that the service will always be in a consistent state if their conditions were satisfied.

The sets of composition rules, which we propose for each type of complex iteration, such as Sequential, Parallel, Iteration, Repeat-Until, If-Then-Else and Repeat-While must be satisfied by the services that participate in the specific iteration. The rules composed from a set of predicates defined in each constraint (precondition, postcondition and invariant) of the service. Specifically, they determine correlations between the constraints of the individual Web services. For example, in the sequential composition of two services, the composition rules specify that the postconditions and invariants of the first service, if this has been executed correctly (\( Pre_{first} \land Inv_{first} \land Post_{first} \Rightarrow Inv'_{first} \)), must indulge the preconditions and the invariants of the second service in order for its execution to be efficient (\( Post_{first} \land Inv'_{first} \Rightarrow Pre_{second} \land Inv_{second} \)). The insert of the composition rules in Web service specification languages augments them, with compositionality. Compositionality refers to the technical property that enables reasoning about a composed system on the basis of its constituents parts without any additional need for information about the implementation of those parts. The idea was first formulated by Edsger W. Dijkstra.
Compositionality can also be inserted in Web service specification languages providing reasoning of the basic Web services that participate in the composition by defining rules with predicates. The predicates correlate with the preconditions, postconditions and invariants that have been defined in the specification of the services.

Specifically, the Web service specification languages with the insertion of the composition rules in their definitions do not only constrain on the Web services orchestration but in addition provide verification of the composed services. The orchestration of a Web service determines how the Web services can interact with each other at the messages level, including the business logic and execution order of the interactions. The composed services produced by following these composition rules can determine that their execution will be efficient at the execution time without producing any inconsistent problem between the basic services execution. In addition, considering the composite service as a black box without knowing the execution of the basic services but knowing the complex iteration and the specifications of the basic services only, we can determine the effects that will be produced after the complex service execution. In the below sections, the composition rules that must be satisfied for each type of complex iteration are presented. The types of the composition that we have studied are the following: Sequence, Parallel, Iteration, Repeat-Until, If-Then-Else, Repeat-While and Choice.

### 3.1 Sequential Composition

The Web services that participate in a sequential composition are executed in a specific row. In order to start the execution of a service, the service that is executed in this particular time must have been completed. The execution of a composite Web service depends on the execution of the individual Web services that participate in the composition. We make the assumption that in the initial state, all the invariants are satisfied, so the execution of a Web service starts in a consistent situation. This condition refers only to the initial state, because a composite service can pass through many inconsistent cases, until it reaches the final state. The postconditions and the invariants of a Web service are satisfied, when the service has been executed.
correctly. The next Web service can start its execution only if its preconditions and invariants are satisfied. We can say that the postconditions and the invariants of the previous Web service must entail the preconditions and the invariants of the next executable service. This rule is satisfied only if the set \( S = \{ \text{Post}_i, \text{Inv'}_i, \text{Pre}_{i+1}, \text{Inv'}_{i+1} \} \) is consistent; this means that there must be an interpretation that satisfies all the predicates of the set \( S \).

In the following picture it is made obvious that \( i \) th Web services can be composed in sequence with the specific order, if they satisfy the corresponding rules.

![Figure 3-1: Composition rules for sequential composition](image)

The inputs, outputs, preconditions and postconditions of a composite service are specified from the services that take part in the composition. In this kind of composition, the inputs of the complex service are constituted from the inputs of the service that is executed first. In addition, the outputs are specified from the outputs of the last executable service.

![Figure 3-2: Representing the composite service as black box](image)

In order to establish if the sequential composition of a Web service is efficient or not, the condition “\( \text{Post}_i \land \text{Inv'}_i \Rightarrow \text{Pre}_j \land \text{Inv'}_j \)” has to be checked before the execution of any individual Web service.

For the examination of this condition, an algorithm has been developed, that takes into account assumptions and tries to simplify this condition. This algorithm is presented in the following table:
CHAPTER 3. DEFINITION OF COMPOSITION RULES

Vassiliki Alevizou

Inputs:

- Web Service Specification:
  - Web Service 1 ($S_1$): Input$_1$, Pre$_1$, Inv$_1$, Post$_1$
  - Web Service 2 ($S_2$): Input$_2$, Pre$_2$, Inv$_2$, Post$_2$
- Inv$_1$ = true $\land$ Inv$_2$ = true
- Input$_1$ $\land$ Input$_2$ is known

$S_2$ will be executed if ($\text{Post}_1 \land \text{Inv}'_1 \Rightarrow \text{Pre}_2 \land \text{Inv}_2$) == true

- Replace the parameters of the conditions with the inputs of the Web Services
- Use condition Post$_1$ to replace variables that depends on the execution of $S_1$ with the new values
- For the execution of the Web service $S_1$, the Pre$_1$ must be satisfied
- Pre$_1$ = true, so replace any of the conditions, which is satisfied from the condition Pre$_1$ with the value true
- Check the following cases:
  - True $\Rightarrow$ something
    - “something” must be computed
      - If “something” is true
        - Then the composition is able
        - Else the composition is not able
  - False $\Rightarrow$ something
    - This means that $S_1$ was not executed correct. So, $S_2$ must not be executed

The functionality of the above algorithm is presented through the examples below:

Example 1:

In this example, we want to compose in sequence two Web services that withdraw a specific amount of money from the same bank account. The integrity constraints that must be satisfied in the initial and final state of each Web service are presented in the figure below. The parameters A and B correspond to the amount of money that we want to withdraw.

![Figure 3-3: The sequential composition of two withdraw Web services](image-url)

Vassiliki Alevizou
The composition of these Web services will be efficient if the condition 
\( \{ (balance' = balance - A \land balance' \geq 0) \Rightarrow balance' \geq B \land balance' \geq 0 \} \) is satisfied.
This condition can be simplified with the implementation of the above algorithm. The algorithm steps are presented below.

- \( Post_1 \land Inv_1' \Rightarrow Pre_2 \land Inv_2 \)
- \( (balance' = balance - A \land balance' \geq 0) \Rightarrow balance' \geq B \land balance' \geq 0 \)

The conditions that are satisfied from the postcondition (Post1) are used for the replacement of the parameters with the new values after the execution of the Web service S1.

- \( balance - A \geq 0 \Rightarrow balance - A \geq B \land balance - A \geq 0 \)

The condition \( balance - A \geq 0 \) is true because the service S1 can be executed only if the precondition Pre1 is satisfied (Pre1 = true).

- \( True \Rightarrow balance - A \geq B \land True \)
- \( True \Rightarrow balance \geq B + A \)

After the replacements and the assumptions that we made, we simplify the initial condition into the following: “\( True \Rightarrow something \)”. This condition is satisfiable only if the second part of the entailment is true. So, we have to compute only the second part.

It is clear, that the condition that we present does not depend on the executions of the individual services. It is based on the initial state, the Web services specifications and their inputs.

Example 2:
If we want to organize a trip, we have to reserve the tickets transportation and the room for our residence in the specific place. The dates of the tickets reservation must be consistent with the reservation in the hotel. Also, if we cannot find tickets, we don’t want to make a reservation for room. All these constraints can represent the integrity constraints that must be satisfied in order for the composition of two Web services for trip arrangement to be efficient. The trip organization must be made with the use of two Web services, tickets reservation and room reservation, and especially with their sequential composition. Below, the constraints that must be satisfied for each Web service are presented.
As, it is presented in the previous example, this Web services composition will be efficient only if the condition:

\[(ticket\_reservation(x) \land ticket\_availability' = ticket\_availability - x \land total'\_booking \leq limit'\_booking) \Rightarrow Valid\_date \land room\_availability \geq y \land total\_booking \leq limit\_booking\]

is satisfied. The parameters \(x, y\) present the number of tickets and rooms that we want to reserve. With the implementation of the algorithm in this condition, we can simplify it into the following relation:

- \[(ticket\_reservation(x) \land ticket\_availability' = ticket\_availability - x \land total'\_booking \leq limit'\_booking) \Rightarrow Valid\_date \land room\_availability \geq y \land total\_booking \leq limit\_booking\]

The condition (Post1) is used for the replacement of the variables with the new values. So, the new value of the variable ticket\_availability is the old value minus the number of tickets reservations.

- \[total'\_booking \leq limit'\_booking \Rightarrow Valid\_date \land room\_availability \geq y \land total\_booking \leq limit\_booking\]

The statement \(total'\_booking \leq limit'\_booking\) can be computed from the information that we have about the number of the reservation that we can make. After the computation in the first part of the entailment, the statement will take one of the following forms:

- \[True \Rightarrow Valid\_date \land room\_availability \geq y \land total\_booking \leq limit\_booking\]

Or

- \[False \Rightarrow Valid\_date \land room\_availability \geq y \land total\_booking \leq limit\_booking\]

If the first part is false, we don’t have to compute the second part of the entailment and that is because the service \(S_1\) would not have been executed correctly so the \(S_2\) does not need to be executed. But, in the case where the first part is true, we only have to compute the second part. The second part of the entailment contains...
variables that their values can be computed from the information that we have from
the initial state. The variable *Valid_date* is determined by the input of Web service S₁.
Also, the parameters x, y are known from the initial state, so equalities and odds based
on these can be computed. Also in this example, it is obvious that the condition is
based on the Web service specifications, their input and initial state. Knowing this
information, we have the ability to compute it before services execution.

### 3.2 Parallel Composition

In parallel composition all the services that participate may be executed
independently. The services can be executed concurrently. In both cases, the
composition will be efficient if in the initial state the preconditions and the invariants
of all the services components are satisfied. As the services are executed
independently, the invariants of each of them must not be inconsistent with the
invariants of the rest services. It is obvious that the set, which is determined from the
services invariants, must be consistent, so that the execution of any service does not
produce problems to other services. In the end, when all the services have completed
their executions, the postconditions and the invariants of all of them must be held. The
rules analyzed above are presented in the following picture.

![Figure 3-5: Composition rules for parallel composition](image_url)
Before, the execution of each Web service that participates in parallel composition with others, we have to check if the above rules are satisfied. These rules can be simplified with the use of the following algorithm.

- **Inputs:**
  - Web Service Specification:
    - Web Service 1 (S₁): Input₁, Pre₁, Inv₁, Post₁
    - Web Service 2 (S₂): Input₂, Pre₂, Inv₂, Post₂
  - Inv₁ = true \(\land\) Inv₂ = true
  - Input₁ \(\land\) Input₂ is known
- S₁ || S₂
- Check first if \((\text{Pre₁} \land \text{Inv₁} \land \text{Pre₂} \land \text{Inv₂})\) is true
  - Replace the parameters of the conditions with the inputs of the Web Services
  - Inv₁ = true, so replace any of the conditions, which is satisfied from the condition Inv₁ with the value true
  - Inv₂ = true, so replace any of the conditions, which is satisfied from the condition Inv₂ with the value true
  - If \((\text{Pre₁} \land \text{Inv₁} \land \text{Pre₂} \land \text{Inv₂})\) is true
    - Check \((\text{Post₁} \land \text{Inv₁'} \land \text{Post₂} \land \text{Inv₂'})\) (1)
      - Replace the parameters of the conditions with the inputs of the Web Services
      - Use condition Post₁ to replace variables that depends on the execution of S₁ with the new values
      - For the execution of the Web service S₁, the Pre₁ must be satisfied
        - Pre₁ = true, so replace any of the conditions, which is satisfied from the condition Pre₁ with the value true
      - Use condition Post₂ to replace variables that depends on the execution of S₂ with the new values
      - For the execution of the Web service S₂, the Pre₂ must be satisfied
        - Pre₂ = true, so replace any of the conditions, which is satisfied from the condition Pre₂ with the value true
      - Check the following cases:
        - If (1) is true
          - the composition is able
        - Else the composition is not able
      - Else if \((\text{Pre₁} \land \text{Inv₁} \land \text{Pre₂} \land \text{Inv₂})\) is false the composition is not able
      - Else if \((\text{Pre₁} \land \text{Inv₁} \land \text{Pre₂} \land \text{Inv₂})\) is unstable, we cannot ensure for the composition

*Example:*

Supposing, that a user wants to make online transactions using the same credit card. Each transaction will be completed if its conditions are satisfied. In addition, the execution of these transactions can be executed concurrently only if the balance of the
credit card is enough for all the shopping. The picture below shows the conditions that must be satisfied for the shopping of a book and a CD. First of all, these two things must be available and also the credit charge must not be greater than 400 dollars. During the execution of the specific transactions the credit charge must be kept less than 400 dollars. Finally, when the two transactions will be completed, the amount with which we can charge the credit would be decreased.

![Figure 3-6: Parallel composition of online transactional Web services](image)

3.3 Iterational Composition

In iterational composition, Web services are executed for a number of times. We can consider this composition as a sequential composition of a Web service with itself. The service will start being executed only if its preconditions and invariants are satisfied. Thus, a service which has been executed for at least once can be executed again only if the preconditions and invariants are indulged in the new produced values. In the final state the postconditions and the invariants of this service are satisfied. The service can be executed for another time only if the following rule is satisfied: \[ \text{Post}_i \land \text{Inv}_i \Rightarrow \text{Pre}_{i+1} \land \text{Inv}_{i+1} \]. This condition must be checked every time the service terminates, and that's because the conditions (preconditions & postconditions) may not be based explicitly on the number of the iterations. Supposing that we have a postcondition that increases a variable with a constant value and that the precondition specifies a check on this specific variable, in this case we have the ability to determine before service’s execution if the execution of the service
is efficient for a number of iterations. In the case where the variable is increased with a random number, we don’t have the ability to verify from the beginning if the execution of the service for the specific number of iterations is applicable or not. Thus, the condition must be verified after the end of each execution.

Example:
Supposing, that we want to withdraw 300 dollars from a bank. In case, that the maximum assumption that we can make is only 100 dollars each time, we must execute this service 3 times in succession in order to withdraw 300 dollars. The withdrawal from this bill will be made only if the preconditions and invariants presented in the following picture are satisfied.

Each time, we have to check if the balance of the specific bill is equal or greater than the amount of 100 dollars \((balance \geq 100)\). This means that each time a withdrawal finishes and before a new one starts, we must check if the new balance that arises follows the preconditions and the invariants of the service \((balance' = balance -100 \& balance' \geq 0)\). Specifically, we have to check if the postconditions imply the preconditions of the service. This examination can be made with the use of the following algorithm.
Web Service Specification:

- Web Service 1 (S): Input, Pre, Inv, Post
- Inv = true
- Input is known

Check \((\text{Pre} \land \text{Inv} \land \text{Post} \Rightarrow \text{Inv'})\) (1)

- Replace parameters of the conditions with the inputs of the Web services
- Inv = true, so replace any of the conditions, which is satisfied from the condition Inv with the value true
- If \((\text{Pre} \land \text{Inv} \land \text{Post} \Rightarrow \text{Inv'})\) is false, then the composition is not able
- Else if \((\text{Pre} \land \text{Inv} \land \text{Post} \Rightarrow \text{Inv'})\) is unstable, we cannot ensure for the composition
- Else if \((\text{Pre} \land \text{Inv} \land \text{Post} \Rightarrow \text{Inv'})\) is true

While \((i < n)\): number of iteration

- S has been executed. So \(\text{Pre} \land \text{Inv} \land \text{Post} \Rightarrow \text{Inv'}\) (Inv' = true)
- Check \((\text{Post} \land \text{Inv'} \Rightarrow \text{Pre'} \land \text{Inv'})\)
- If \text{Pre'} contains variables that their values depend on postconditions, replace the variables with the new values.
- Then check the condition
- If \((\text{Post} \land \text{Inv'} \Rightarrow \text{Pre'} \land \text{Inv'})\) is true
  - Then S is executed for another time. So \(\text{Pre'} \land \text{Inv'} \land \text{Post'} \Rightarrow \text{Inv''}\)
  - And Inv'' = true & i++
- Else execution terminates

### 3.4 Repeat-Until Composition

With the operation Repeat-Until, the Web service that participates in the composition will be executed for the first time, only if its conditions are satisfied. In order to continue its execution, the condition G must not be satisfied. In addition, the conditions of the Web service in the new computed values must be satisfied in order for the service to be executed for another time. In other words, the negation of the condition G must entail the conditions for the correct execution of the service. Thus, the following rule: \(¬G \Rightarrow ((\text{Pre'} \land \text{Inv'} \land \text{Post'}) \Rightarrow \text{Inv''})\) must be satisfied.

\[
¬G \Rightarrow ((\text{Pre'} \land \text{Inv'} \land \text{Post'}) \Rightarrow \text{Inv''}) \equiv \\
¬G \Rightarrow ¬(\text{Pre'} \land \text{Inv'} \land \text{Post'}) \lor \text{Inv''} \equiv \\
¬G \Rightarrow ¬\text{Pre'} \lor ¬\text{Inv'} \lor ¬\text{Post'} \lor \text{Inv''} \equiv \\
¬G \lor \neg \text{Pre'} \lor \neg \text{Inv'} \lor \neg \text{Post'} \lor \text{Inv''} \equiv \\
G \lor \neg \text{Pre'} \lor \neg \text{Inv'} \lor \neg \text{Post'} \lor \text{Inv''} \equiv \\
\]
Vassiliki Alevizou

\[ \neg(-G \land Pre' \land Inv' \land Post') \lor Inv'' \equiv \neg G \land Pre' \land Inv' \land Post' \Rightarrow Inv'' \quad (1) \]

The above entailment (1) presents that a Web service can be executed again only if this condition is verified. If the predicates of the first part of the entailment are true, then the invariants that refer to the new values of the variables must also be satisfied. Thus, the set that consisted from the entailment predicates must be consistent. This means, that there is an interpretation that confirms all these predicates.

![Figure 3-9: Composition rules for Repeat-Until composition](image)

The entailment \((\neg G \land Pre' \land Inv' \land Post' \Rightarrow Inv'')\) can be computed with the use of the following algorithm. The algorithm must be applied every time the service finishes its execution and before it’s executed for another time.

- **Inputs:**
  - Web Service Specification:
    - Web Service (S): Input, Pre, Inv, Post
  - Inv = true
  - Input is known
  - Condition G
  - S will be executed for the first time if (Pre \land Inv \land Post \Rightarrow Inv) is true
  - Replace the parameters of the conditions with the inputs of the Web services
  - Inv = true, so replace any of the conditions, which is satisfied from the condition Inv, with the value true
  - If (Pre \land Inv \land Post \Rightarrow Inv') is false, then the composition is not able
  - Else if (Pre \land Inv \land Post \Rightarrow Inv') is unstable, we cannot unsure for the composition
  - Else if (Pre \land Inv \land Post \Rightarrow Inv') is true
    - So, Inv' = true
    - S will be executed for another time if (\neg G \land Pre' \land Inv' \land Post' \Rightarrow Inv'')

Vassiliki Alevizou
Use Post to replace the variables in condition Pre' with the new values
Use Post' to replace the variables in condition Inv'' with the new values
Replace the parameters with the inputs of the Web services
Check if any of the conditions is satisfied from the Inv' which is true
Check the following cases:
  ♦ True ⇒ something
    ➢ “something” must be computed
      • If “something” is true
        • Then, S can be executed for another more time
        • Else, the execution of S terminates
  ♦ False ⇒ something
    ➢ S cannot be executed again

Example:

Supposing that, we have a service that enrolls students in a specific course. The enrollment of a student is able if the current size of the class does not overdose the maximum number of students. This condition must be satisfied not only in the initial state of the service but also in the final. Thus, the enrollment of a student should not be completed, if this condition is not satisfied after the student’s enrollment in the specific course. The service can enroll a student only if he is not already enrolled in this course. The service will continue its execution until the following condition “current_size(course) > total_size(course)” becomes true. The constraints that must be satisfied in the initial and final state of the service are presented in the following picture. The execution of the service will be continued if the condition “(current_size'(course) > total_size'(course)) ∧ ¬enrolled(x',course) ∧ current_size'(course)< total_size'(course) ∧ enrolled(x',course) ⇒ current_size''(course)< total_size''(course) ” is satisfiable. This condition can be simplified with the use of the algorithm. With the replacement of the known parameters, the condition is turned in an easier constraint for check. All these conditions are presented in the following picture.
3.5 If-Then-Else Composition

In the “If-Then-Else” composition the selection of the services which will be executed depends on the condition value. Supposing that we have 4 Web services which we want to compose with the following order: $S_1:(S_2 \lor S_3):S_4$. The selection of the services $S_2$ and $S_3$ depends on the value a specific condition (G) has in a particular time. Considering that $S_2$ is executed when condition G is true and $S_3$ when G is false.

Initial, $S_1$ will start its execution if its preconditions and invariants are satisfied. After the execution of $S_1$, the value of the condition G for the selection of the next service must be checked. The $S_2$ will be applicable for execution when the condition G, the preconditions and the service’s invariants are satisfied. Thus, the following condition must be satisfied:

\[
G \Rightarrow ((\text{Pre}_2 \land \text{Inv}_2 \land \text{Post}_2) \Rightarrow \text{Inv}_2') \equiv \\
G \Rightarrow (\lnot(\text{Pre}_2 \land \text{Inv}_2 \land \text{Post}_2) \lor \text{Inv}_2') \equiv \\
G \Rightarrow \lnot\text{Pre}_2 \lor \lnot\text{Inv}_2 \lor \lnot\text{Post}_2 \lor \text{Inv}_2' \equiv \\
\lnot G \lor \lnot\text{Pre}_2 \lor \lnot\text{Inv}_2 \lor \lnot\text{Post}_2 \lor \text{Inv}_2' \equiv \\
\lnot(G \land \text{Pre}_2 \land \text{Inv}_2 \land \text{Post}_2) \lor \text{Inv}_2' \equiv \\
G \land \text{Pre}_2 \land \text{Inv}_2 \land \text{Post}_2 \Rightarrow \text{Inv}_2' \quad (1)
\]

The relation (1) presents that the set $S= \{G, \text{Pre}_2, \text{Inv}_2, \text{Post}_2, \text{Inv}_2'\}$ must be consistent. In contrast, the service $S_3$ will be executed when the condition G is not
satisfied but its preconditions, invariants and postconditions are. So, the following condition must be verified:

\[-G \Rightarrow ((\text{Pre}_3 \land \text{Inv}_3 \land \text{Post}_3) \Rightarrow \text{Inv}'_3) \equiv\]

\[-G \Rightarrow (\neg(\text{Pre}_3 \land \text{Inv}_3 \land \text{Post}_3) \lor \text{Inv}'_3) \equiv\]

\[-G \Rightarrow \neg\text{Pre}_3 \lor \neg\text{Inv}_3 \lor \neg\text{Post}_3 \lor \text{Inv}'_3 \equiv\]

\[-G \lor \neg\text{Pre}_3 \lor \neg\text{Inv}_3 \lor \neg\text{Post}_3 \lor \text{Inv}'_3 \equiv\]

\[-(\neg G \land \text{Pre}_3 \land \text{Inv}_3 \land \text{Post}_3) \lor \text{Inv}'_3 \equiv\]

\[-G \land \text{Pre}_3 \land \text{Inv}_3 \land \text{Post}_3 \Rightarrow \text{Inv}'_3 \quad (2)\]

As it is referred, the condition G will determine the selection between the services S₂ and S₃. The service S₄ will be executed when the previous service has been terminated. Depending on the previous service, one of the following set must be consistent: S: \{\text{Post}_2, \text{Inv'}_2, \text{Pre}_4, \text{Inv}_4\}, S': \{\text{Post}_3, \text{Inv'}_3, \text{Pre}_4, \text{Inv}_4\}. In the end of all services execution only the postconditions and the invariants of the service S₄ are satisfied.

The examination of the validity of the entailments (1), (2), can be made with the use of an algorithm.
Example:

Supposing, that we want to organize a trip in a city. Besides the room reservation, we want to rent a car if the hotel is on the outskirts of the city, otherwise we want to rent a bike. First, in order to execute the service for the hotel reservation, its preconditions and invariants must be satisfied. When the room reservation has been completed, depending on the hotel location we will execute either the service for renting a car or the service for renting a bike. This constraint comprises the condition in the If – Then – Else operation. Besides the condition, for the execution of each service the corresponding constraints for each of them must be satisfied. This means
that the condition and the constraints must imply the service execution. This condition is presented in the following picture with bold letters.

![Diagram showing the If-Then-Else composition of a Rent_Car service and a Rent_Bike service](image)

Figure 3-12: The If-Then-Else composition of a Rent_Car service and a Rent_Bike service

### 3.6 Repeat-While Composition

```plaintext
while (G) {
    execute S
}
```

With the operation \textit{Repeat-While} the service will continue to be executed only if the condition $G$ is determined by the operator “While” will be satisfied. So, each time the validity of the condition must be checked before the execution of the service. After the examination of the condition, the service execution will start only if the preconditions and the invariants of the service are satisfied. So, the entailment that must be examined each time before the service execution is the following ($G \Rightarrow (Pre \land Inv \land Post \Rightarrow Inv)$).
The above entailment (1) determines that besides the condition satisfiability must also satisfy the conditions of the services in order to start its execution. In addition if the predicates of the first part of the entailment are satisfied then the invariants of the new values of the variables must also be satisfied. Specifically, the execution of the service must be correct. In other words, the set that is composed from the predicates of the entailment must be consistent. The consistence of the set means that there is an interpretation which confirms all these predicates.

![Figure 3-13: Composition rules for the Repeat-While composition](image)

The algorithm which can be used for the entailment simplification is presented below:

- **Inputs:**
  - Web Service Specification:
    - Web Service (S): Input, Pre, Inv, Post
  - Inv = true
  - Input is known
  - Condition G
- **S** will be executed while \((G \land Pre \land Inv \land Post \Rightarrow Inv')\)
  - Inv = true
  - Use Pre and inputs in order to produce the postconditions that will be
produced after service execution.

- Use Post to replace the variable in condition Inv’ with the new values
- Replace the parameters with the inputs of the Web services
- Check if any of the conditions is satisfied from the Inv which is true
- Check the following cases:
  - True ⇒ something
    - “something” must be computed
      - If “something” is true
        - Then, S can be executed for another more time
        - Else, the execution of S terminates
  - False ⇒ something
    - S cannot be executed again

Example:

Let’s consider that we have a Web service that enrolls students in a specific course. In this course only the students that have passed the course “hy140” and have taken the course for the first time can be enrolled. The service will be found in a consistent state only if the total number of the students of the specific school is smaller than the number of the students that can study in this school.

Due to the limited number of students that can participate in a specific course, the service can be executed as many times as the number of the students that can be enrolled. The preconditions, postconditions, and invariants of the service are presented in the following figure.

![Figure 3-14: The Repeat-While composition of a Web service for enrolling students in a course](image-url)
3.7 Choice

Supposing that we have a set \( S = \{S_1, S_2, \ldots, S_n\} \) of services. The choice of \( m \) services from the set \( S \) with \( n \) services is a non-deterministic choice.

\[
\text{choice}(C) = C \subseteq S : |C| = m
\]

where, \( C \) is a set that contains \( m \) services from the \( n \) services of the set \( S \).

For the efficient composition of the \( m \) selected services (set \( C \)) from the set \( S \), the services must satisfy the rules for each complex iteration.

- **Sequential Composition**
  Let \( C \) consists of the following services, \( C = \{S_1, S_2, \ldots, S_m\} \). These services can be composed in sequence if they satisfy the rules presented in the section 3.1. These rules depend on the order that the services participate in the composition.

- **Parallel Composition**
  The same must be satisfied for the parallel composition. So, the services that composed the set \( C = \{S_1, S_2, \ldots, S_m\} \) must satisfy the appropriate rules.

- **Iterational Composition**
  After the selection of a service from the set \( S \) of Web services, the rules that we define in the paragraph about the iterational compositional must be checked.

- **Repeat-Until Composition**
  The same must be followed for the Repeat-Until composition. The service selected has to satisfy the corresponding rules for services selection until a condition becomes true.

- **If-Then-Else Composition**
  In the If-Then-Else composition, after the selection of the services, the appropriate rules for the specific composition must be satisfied.

- **Repeat-While Composition**
  In the Repeat-While composition, the service which will be selected must satisfy the corresponding rules in order to start its execution. This condition must always be checked before the service is executed for another time.
Chapter 4

Augment backward approach planners with composition rules

In the previous chapter, are being introduced the composition rules that must be satisfied between the services which participate in a specific type of iteration. These rules determine the effects of a composite service. Specifically they entitle the properties of a composite service such as inputs, outputs, preconditions, postconditions and invariants from the inputs, outputs, preconditions, postconditions and invariants of the sub-services. These rules can be used for the augmentation of a planner of the backward based approaches planners due to their ability to enable reasoning for service selection and composition.

In the section 4.1 we study the goal formalization, how the goal can be presented in the method of AI planning. We consider two cases for Web service composition in section 4.2. Firstly, we present the insertion of the composition rules in the case where an ontology is provided with the complex goal. Secondly, in the subsection 4.2.2 we analyze the insertion of the composition rules in the backward approaches planners. A general algorithm for the plan generation is provided and a comparison between the Depth first and Breadth first approaches is made. Finally, we present a set of provers for first-order logic that can be used for the satisfiability of the composition rules.

4.1 Goal Formalization

In an environment of semantic Web services, in order to achieve certain goals users could use software agents which automatically identify the user’s needs and if it is necessary compose services in order to accomplish these goals. However, the problem of dynamic composition can be described as a hard problem until now and it is not
always clear which technique can be used for the problem solving. There are many works that have been proposed for dynamic composition. One of them is the AI (Artificial Intelligence) planning. A planning problem has the following components:

- A description of the possible actions which may be executed.
- A description of the initial state of the world.
- A description of the desired goal (user’s needs)

There are many approaches that can be used for the definition of the components of the planning problems. These types of definition determine the approach that will be used for the Web service composition problem.

In most classical approaches, the goals can be presented as a set of properties that must be satisfied in a specific desired world state. Usually, these properties have a form of conjunctions or disjunctions of literals, which contain positive or negative atoms. This approach of goal presentation is not the most sufficient for the automatic Web service composition. In the case where the goals are complex, specifically if they have been defined with branching and loops, the goal must be divided into several distinct parts in order to define each of them separately. In addition, when specifying the goals as properties we cannot take into consideration the preferences that a user may have. In addition, goals can be specified with the use of STRIPS. In STRIPS the goal is specified as a set of properties that must be satisfied after the execution of a set of actions.

Moreover, a goal can be represented as a situation $s \ (d_0(\bar{a}, S_0))$ ([MS02]) that can be achieved after the execution of a sequence of actions $(\alpha_1, \alpha_2, \ldots, \alpha_{n-1}, \alpha_n)$ in a specific state $(S_0)$. The state of the world is described by functions and relations relativized a situation $s$, e.g., $f(x, s)$. Specifically, given a situation calculus theory $D$ and a Golog program $\delta$, program execution must find a sequence of actions $\alpha$ such that: $D \models D_0(\delta, S_0, d_0(\bar{a}, S_0))$. Where, the $D_0(\delta, S_0, d_0(\bar{a}, S_0))$ denotes that the Golog program $\delta$ starting execution in $S_0$ will legally terminate in situation $d_0(\bar{a}, S_0)$.

Also, a service composition problem can be described as a planning problem in PDDL. The specification of the Web services with the PDDL provides them with semantic markup. With the use of this language a high-level description of a task can
be formulated as a simple goal. The PDDL specification of a goal and of a planning domain is supported by a wide range of planners.

Thus, for each of these types of goal specifications a set of planners has been proposed that can be used for the resolution of the Web service composition problem.

4.2 **Composing Web Services**

Considering a composition problem as a planning problem, there have been a lot of studies for the generation of the most efficient planner. In general, the planning problem can be described as a challenging problem, in which each study tries to resolve this in a more efficient way. The kind of planner that we have to select is based on the information that we have in our disposal as well as on the domain that used for the goal and the initial state specification.

4.2.1 **Providing an ontology for composite service**

As the most ideal case for which we have to generate a plan in order to solve a problem we can consider the case that besides the set of the services that can be used in composition and the initial state, there is also an ontology available that represents the complex Web service.

In this case, the user has an ontology in his disposal that represents the relation between a complex goal and the services that must be used for the goal materialization. These services are defined as subclasses of the complex service in the ontology definition. In addition, ontology presents all the possible combinations of the main services, the execution of which indulges the goal. Specifically, the number of these types is not equal with the number of all the alternative combinations of the available services, but only with the combinations, the execution of which under some circumstances provides the desired goal. These types are presented as subclasses of the class “Composition”. All these classes have Web services as instances that were defined in the same specification language. For example, the services which
participate in the composition of the goal have all the appropriate Web services as instances that provide the specific type of service.

The classes, which determine the effective types of composition, have the specification of a complex Web service as instances, which are produced from the composition of the services instances. The type of composition is determined by the class in which the instance belongs. These specific instances have been produced, considering the composition rules that must be satisfied in each type of iteration. So, the execution of these instances is effective, if the initial state satisfies the inputs, and the preconditions of the composite service. Each of these instances comprises a plan of specific services execution. Considering the above, the initial state plays the decisive role in plan selection. The plan, which will finally be selected, will be that in which the constraints and the information of the initial state satisfy the constraints and the inputs of the plan, in other words, the specification of the complex Web service.

In the below figure an ontology of a complex service that organizes a trip “Travel service” is presented. The main services that must be used for the composition of this specific service are the following: Air booking ($S_1$), Room reservation ($S_2$), and Car leasing ($S_3$). The efficient types of composition are presented as subclasses of the class “Composition”. If we can consider that all the sequential combination of these three services can give us the desire goal and follow the composition rules, we have to define 27 alternative kinds of composition as subclasses of the class “Composition”. Each of them has two instances in accordance with the available instances of the main Web services. Thus, we have to check 54 plans in order to see which of them is satisfied by the initial state.
In this ontology, we consider that all the main services which are defined as subclasses of the goal must be used in goal specification. With this assumption, we reduce the number of the alternative plans that we have to check, however, we constrain the plans that can be covered by this ontology.

In this approach, the user does not have the ability to insert constraint in the plan production, and that is because the ontology is fixed and the plans are stated. The only thing that the user can do is to select the most relevant plan in his preferences. This approach is the most ideal we can have. It is very unusual to have an ontology in our disposal that represents the efficient types of composition that indulge a specific goal.

### 4.2.2 Composite Web services without an ontology providing

However, in dynamic composition, the occurrence of an ontology that specifies the composed goal is rare. Usually, a complex goal is provided from a user and an agent tries to solve it with the execution of a basic service or the composition of a set of services. The only thing that is known is a set of information which must be indulged in the initial state. The services that can be used for the goal generation are selected from a repository of available services. Considering this information, a plan must be found, the execution of which satisfies the specific complex goal. As it has been referred, the works that have been proposed, implement a set of planners which can be
used for plan generation considering the initial state, the available services and the desired goal.

In most proposed works, the selection of the services that will be used for the plan generation is based on satisfaction of the services properties and conditions such as inputs, outputs types and on the kind of their preconditions and postconditions. For example in the case of a planner that generates a plan with the approach of goal decomposition, in each step it select services where their outputs and postconditions can indulge the inputs and the preconditions of the services of the above level. The correlation is based only on the properties matching. In ([desFS03]) is assumed that after a service execution only the effects that are desired for the composition will always be satisfied. So, the execution of the next service that depends on the effects of the previous Web service will be efficient. In addition some other approaches use the procedure of reasoning in order to prove the evaluation of the services preconditions. Some planners are provided with the ability to execute information-gathering services in order to satisfy specific preconditions. The precondition evaluation using reasoning is able only in the cases where the current state is known.

In accordance to the above, we propose an augmentation of the planners that based on goal decomposition. By decomposition, we mean that a complex goal can be divided into sub goals which can be satisfied by some services execution. The decomposition of the new goals will now be continued until we conclude into a set of services the execution of which is satisfied by the initial state.

The selection of the services will not based only in the semantic correlation that is made between the postconditions of a service and the preconditions of the service that we want to decompose, the planner has a set of rules in its service that must be held between the services in each type of complex composition. The rules represent correlation between the preconditions, invariants and postconditions of the services that have to be composed in a specific way. The producing plan consists of a number of services executed sequentially or parallel. The parallel composition is made when the preconditions of a service demand the execution of more than one service for their satisfaction. We consider that service decomposition can be satisfied by more than one services, so with the use of these rules we can select the most appropriate service for its decomposition, which will not produce any inconsistent
problem after its execution. Specifically, we ensure that the execution of a service will not produce any constraint which prohibits the execution of the next service.

All the composition rules used for the services selection are proven with the use of a suitable theorem prover. Considering a set of axioms and a knowledge base, the prover will determine if a specific composition rule among a set of services is satisfiable or not. Specifically, the prover will determine if a specific service can be composed sequential or parallel with another service or set of services, respectively. The knowledge base will consist of the properties of services which are satisfied such as preconditions, invariants and postconditions. The axioms will be comprised from the rules satisfied by the correct execution of the services. The most important thing that must be made clear is that the planner that will generate a plan will be sound and complete, if the theorem prover used for the proof of the rules satisfiability is sound and complete. And that is because the composition rules among the services of a specific plan will be satisfied, so the planner will always produce a plan, if there exists one and it will always be the correct for the goal satisfaction. The introduction of these composition rules in the planners augments the reasoning power for Web service composition.

We should point out that the composition rules take into consideration all the possible postconditions that can be produced after a service execution. It is known that each Web service can have conditional postconditions. These take place only if their respective conditions are satisfied after the service execution. Thus, with the use of satisfiability theorem provers for the rules, we ensure that the services composition will be efficient without precluding the postconditions that must be satisfied after the services execution.

### 4.3 Logical Form for Goal Specification

The goal is encountered as a set of postconditions that must be satisfied after a specific plan execution. Each service produces a set of postconditions after its execution. These postconditions will be used for the selection of the most appropriate service for the decomposition of a specific one. The graph representation of a plan
with the correlation between the postconditions and the preconditions of the services that participate in it is presented below.

![Figure 4-2: A plan with correlations between the preconditions and postconditions](image)

The services of the same level can be executed in parallel, whereas the services of different levels are executed in sequence with a reversing order of that of the decomposition. For example the services $S_3$ and $S_4$ can be executed in parallel, but the $S_1$ has to wait the termination of the $S_3$ in order to start its execution. The rules that we have to check for the decomposition of a service into another subservices are presented in the below figure. In this figure, we show how the composition rules are used for services decomposition.

![Figure 4-3: A plan and the insertion of the composition rules](image)
The conditions of the Web services, preconditions, postconditions and invariants can be represented efficiently in the first-order logic. A formulae in first-order logic may contain boolean connectives and also variables \( x, y, z, \ldots \), predicates \( P, Q, R, \ldots \), function symbols \( f, g, h, \ldots \), and quantifiers \( \forall \) and \( \exists \) meaning “for all” and “there exists”. Formulae specified in first-order logic may contain constant symbols like \( a, b \) which can be regarded as functions of no arguments. So, as the goal consisted of the postconditions of the services that have been executed, it will be presented as a formulae specified in first-order logic.

The specific goal specification enables us with the efficient use of the compositions rules and the theorem provers for the rules satisfiability proving. We indicate that the goal must be presented in normal form, so we select the conjunctive formal form (CNF) for goal representation. A predicate may be positive \( P \) or negative \( \neg P \). Each formula specified in the first-order logic can be converted in the conjunctive normal form. As the goal presented in the CNF form, it will consist from conjunctions of disjunctions. In general, the goal will have the following form: \( (A \lor B) \land (C \lor E) \).

The Web services which will satisfy the predicates of the clauses \( (A \lor B) \) and \( (C \lor E) \) must follow the rules for the parallel composition. While, if the predicates of each of these clauses need to be satisfied by the execution of more than one service, these services do not need to follow the parallel composition rules. As the clauses consisted of disjunctions of predicates, this means that at least one of these predicates must be satisfied. So, at least one of the services must be executed they can be executed.

A general algorithm which can be used for the plan generation based on goal decomposition is presented below:

**Step1** Considering the \( g \) as a set of predicates, try to find services the postconditions and the invariants \( (Posti \land Invj) \) of which indulge the predicates of the clauses. Specifically, try to find services that indulge the sequential composition rules.

**Step2** Among the services that joined with the conjunction operator check if the parallel composition rules are satisfied. When the predicates are joined with the disjunction operator, it means that at least one of them must be satisfied. So, the services that satisfy those predicates after their execution need not to be checked for the satisfiability of the parallel composition rules.

Vassiliki Alevizou
However, the conditions of these services participate in the parallel composition rules among all the rest services.

**Step 3** If Step 2 returns true, it produces a new goal from the preconditions and the invariants $P_{e_{i}} \land I_{n_{i}}$ of the new selected services $g^{'} = \{C_{1}^{'} \land C_{2}^{'} \land \ldots \land C_{i}^{'}\}$ where now $C_{i}^{'} = \{(P_{e_{i}}^{'} \land I_{n_{i}}^{'})) \lor \ldots \lor (P_{e_{ii}}^{'} \land I_{n_{ii}}^{'}))\}$

**Step 4** If Step 2 returns false, the algorithm terminates with failure.

**Step 5** If there are preconditions and invariants that can be satisfied from the initial state then $g = g^{'} - \{P_{e_{i}}, I_{n_{i}}\} \in initial$ state.

**Step 6** If $g = \emptyset$, then terminate with success.

**Step 7** Else, goto Step 1

### 4.3.1 Depth and Breadth first Approach for plan generation

For the plan generation we have made the assumption that every service can be decomposed into more than one services or more than one set of services. So, during the plan generation and the decomposition of a specific service, we have to select one of the efficient services for its decomposition.

The availability of more than one service for the decomposition of a service means alternative plans for the satisfaction of a specific goal. The alternative plans can be generated one each time or all of them together. So, we can apply two different approaches for plan producing. These two approaches are based on the depth first search and breadth first search approach. Below, the general steps that must be followed in each of these two approaches are presented.

#### 4.3.1.1 Depth First Approach

In the depth first approach, the planner tries to construct only one plan at a time. If there is more than one service for a specific service decomposition which satisfies the sequential composition rules, it selects one of them and continues with the execution.
of the next steps of the algorithm. In the case where the algorithm cannot find a plan with the choices made, we follow the method of the backtracking. So, one of not already used services is selected and the algorithm continues to be executed again. The steps of the planner following the depth first approach are presented below:

**Step1** Considering the $g$ as a set of predicates, for each predicate try to find services the postconditions and the invariants $(Post_i \land Inv_i)$ of which the predicates $(i=1$ and $S_{Selected}=\emptyset)$ indulge. Specifically, try to find services that indulge the sequential composition rules. In the first time that the algorithm will be executed, the $g$ will consist of a set of clauses $(C_1 \land C_2 \land \ldots \land C_i)$.

**Step2** If there are more than one service that can be used for a predicate satisfaction, $S=\emptyset$, $L=L+1$ produce all the alternative sets with the services, the execution of which satisfies the predicates of the previous level and selects one of them $S_L = \{S_1, S_2, \ldots, S_{Last}\}$ for $1 \leq i \leq Last$. $S_{Selected} = S_i$

**Step3** Else, if there are no services that can be used for the predicates satisfaction $g\neq\emptyset$ and $S\neq\emptyset$ and $i\leq Last$ then $i++$ and $S_{Selected}=S_i$ (backtracking).

**Step4** Else, if there are no services that can be used for the predicates satisfaction $(g\neq\emptyset)$ and $S=\emptyset$ or $g\neq\emptyset$ and $i> Last$ then backtracking into the above level and select the next set of services,

**Step5** if there is not a previous level then the algorithm terminates with failure.

**Step6** Else, if there is previous level select the next set of services and goto to **Step8**.

**Step7** Else, if there is previous level, but there are not other set of services goto **Step4**.

**Step8** Among the services $(S_{Selected})$ that joined with the conjunction operator check if the parallel composition rules are satisfied. When the predicates are joined with the disjunction operator, it means that at least one of them must be satisfied. So, among the services that satisfy, those after their execution do not have been checked for the satisfiability of the parallel composition rules.

Vassiliki Alevizou
However, the conditions of these services participate in the parallel composition rules among all the other services.

**Step 9** If **Step 8** returns true, it produces a new goal from the preconditions and the invariants $Pre_i \land Inv_i$ of the new selected services $g' = \{C_1' \land C_2' \land \ldots \land C_i'\}$ where now $C_i' = \{(Pre_{ii} \land Inv_{ii}) \lor \ldots \lor (Pre_{ii} \land Inv_{ii})\}$. Go to **Step 16**.

**Step 10** If **Step 8** returns false (backtracking),

**Step 11** $i++$ if $i \leq \text{Last}$ then $S_{\text{Selected}} = S_i$ and goto **Step 8**

**Step 12** Else if $i > \text{Last}$ then backtrack into the above level ($L=L-1$) if there is above level.

**Step 13** If there are more set of services, selected the next set $S_{\text{Selected}} = S_{i++}$. Goto **Step 8**.

**Step 14** Else, execute again **Step 12**.

**Step 15** Else, terminate and return failure.

**Step 16** If there are preconditions and invariants that can be satisfied from the initial state $g = g' - \{Pre_i, Inv_i\} \in \text{initial state}$

**Step 17** If $g \neq \emptyset$, execute **Step 1**.

**Step 18** Else, return success

**Example:**

Consider that we want to generate a plan for the satisfaction of a goal $A \land B$. This plan will consist of a set of services the execution of which will produce the complex goal. We suppose that the predicate $A$ can be satisfied by the postconditions of the service $S_1$, while the predicate $B$ can be satisfied by the execution of two services $S_2$ and $S_3$. Thus the $S_1, S_2$ and $S_1, S_3$ comprise two alternative sets for the decomposition of the goal $A \land B$. Applying the depth first approach for the plan generation, we have to select one of these two sets and continue the algorithm execution on the selected set. Supposing that it is selected the set $S_1, S_2$ (figure 4-4(a)). The preconditions and invariants of these two services are not satisfied by the initial state, so we will continue with the decomposition of the goal $Pre_1 \land Pre_2$. These predicates can be satisfied with the parallel execution of the services $S_4$ and $S_5$. For the execution of these services, the conditions $Pre_4$ and $Pre_5$ must be satisfied. However,
the initial state cannot provide information for these predicates satisfiability, so we have to find a set of services the execution of which will satisfy them.

Supposing that the services cannot be decomposed into a new set of services, we have to backtrack (figure 4-4(c)) and to check if there is another set of services that can be used for the satisfaction of goal $Pre_1 \land Pre_2$. In the case where there is not another set, we have to backtrack in the above level and find another set of services for the specific goal level. In our case, the goal is the $A \land B$. We showed in the start that these predicates can be produced not only with the execution of the services $S_1, S_2$ but also with the $S_4, S_5$. Thus, now the second set is selected $S_4, S_5$. The same procedure is now applied for the new set. The conditions of these services $Pre_1 \land Pre_3$ can be indulged by the parallel composition of the services $S_4$ and $S_6$, the conditions of which are satisfied by the initial state. Thus, the algorithm terminates with success and will return the following plan: $(S_4 \parallel S_6) : (S_1 \parallel S_3)$.

![Figure 4-4: A Depth first approach for the generation of a plan for the goal: A $\land$ B](image)

4.3.1.2 **Breadth First Approach**

However, in the breadth first approach, the planner will construct all the alternative plans simultaneously. So, it applies all its steps in all the alternative services of specific service decomposition. In contrast to the depth first approach, it does not select any of the alternative sets for the goal satisfaction but it implements all its steps
in all the alternative sets of services. Generally, the planner will apply the following steps:

**Step 1** For each set $g$, for each predicate of this set, try to find services where the postconditions and the invariants $(Post_i \land Inv_i)$ indulge the predicates. Specifically, try to find services that indulge the sequential composition rules.

**Step 2** If there is more than one service that can be used for a predicate satisfaction, then produce all the alternative sets of services, the execution of which satisfy the predicates of the previous level.

**Step 3** For each set of services, we have to check if the parallel composition rules are satisfied. For the services that satisfy the predicates and they are joined with the conjunction operator, check if the parallel composition rules are satisfied. When the predicates are joined with the disjunction operator, it means that at least one of them must be satisfied. So, the services for these predicates satisfaction don’t have to be checked for the satisfiability of the parallel composition rules.

**Step 4** If **Step 3** returns true, produce a new goal from the precondition and invariants $Pre_i \land Inv_i$ of the new selected services $g' = \{C'_1 \land C'_2 \land ... \land C'_i\}$ where now $C'_i = \{(Pre_{i1} \land Inv_{i1}) \lor ... \lor (Pre_{ii} \land Inv_{ii})\}$.

**Step 5** If there are preconditions and invariants that can be satisfied by the initial state, then $g = g' - \{Pre_i, Inv_i\} \in$ initial state.

**Step 6** If $g \neq \emptyset$, then $G = G \cup g$, where $G$ contain all the alternative sets of predicates for the specific level.

**Step 7** Else if $g = \emptyset$, the algorithm terminates and returns success for the specific plan (part of the tree).

**Step 8** Else, if **Step 3** return false, remove this branch from the tree, because the specific set of services cannot be used for the generation of a plan.

**Step 9** Else, if there are no services which can be used for the predicates satisfaction, then this goal is deleted from the set with the goals and the specific branch of the tree is removed.
**Step10** If \( g \neq \emptyset \), then goto **Step1**.

**Step11** Else, the algorithm terminates with failure.

**Step12** If \( G \neq \emptyset \), then \( g = G \) and execute **Step1**.

**Example:**

Consider that we have to generate a plan, the execution of which will satisfy the goal: \( A \land B \). Considering the available services, the predicate \( A \) can be satisfied with the execution of the service \( S_I \). While, the predicate \( B \) can be satisfied with the right execution of two services \( S_2 \) or \( S_3 \). It is obvious that the sentence \( A \land B \) can be satisfied not only with the parallel execution of the services \( S_I \) and \( S_2 \) but also with the parallel execution of the services \( S_I \) and \( S_3 \). The first execution of the above algorithm will produce the tree of the figure (figure 4-5(a)). These two branches of the tree consist two alternative plans that can be used for the goal satisfaction. Having precluding the services in an oval, we determine that these services must be executed in parallel.

We suppose that the services of the sets satisfy the rules for the parallel composition. After the first execution of the algorithm, the goals that we now try to satisfy are the \( Pre_1 \land Pre_2 \) and \( Pre_1 \land Pre_3 \) for each of these two branches, respectively. The \( Pre_1 \), \( Pre_2 \) and \( Pre_3 \) are the conditions that must be satisfied for the correct execution of the services \( S_1 \), \( S_2 \), and \( S_3 \), respectively. With the execution of the algorithm with the new goals, the tree will be converted in the form of the figure (figure 4-5(b)). By checking if the conditions of the new services are satisfied by the initial state, we concluded that the services \( S_4 \) and \( S_6 \) can be satisfied by the initial state. Thus the algorithm terminates with success and with plan: \( (S_4 \parallel S_6) : (S_I \parallel S_3) \).

![](image)

**Figure 4-5:** A Breadth first approach for the generation of a plan of the goal \( A \land B \)

Vassiliki Alevizou
4.3.1.3 Comparing Depth and Breadth first approach

Comparing these two approaches that discussed throughout of the previous sections, we concluded into the following results. In the Depth first approach, each time, we try to generate a possible plan for the satisfaction of a specific goal. Traversing each time a particular branch of the tree, there is a case, in which we have already analyzed this branch in a big depth and we ascertain that the leaves of this branch cannot participate in the plan generation. These leaves can be represented by services that cannot be decomposed anymore and also they cannot be satisfied by the initial state. Thus, valuable time has been spent in this branch decomposition without a result.

However, in the Breadth first approach, we can say that the time for a plan generation is reduced due to the simultaneous examination of all the alternative plans. So, with this approach, the plan can be found faster than the depth first approach and also with the minimum length. As, in the breadth approach the decomposition is applied in all the services of the same level simultaneously, the plan that will be found first will be the smaller in the number of services which participate in it, from the others plans that this tree can contain. Nevertheless, if there are a lot of alternative services that can be used for the decomposition of some services, by applying the algorithm steps in all these services, one after the other valuable time may be lost. Whereas with the depth first approach we may find the plan faster by checking a specific branch of the tree. Concluding, we can say that the breadth first approach is more appropriate in the cases where the alternative services are not many, whereas the Depth first approach is most suitable in the contrary cases.

4.3.2 Theorem Proving in First-Order Logic

During the plan generation, it is obvious that a set of rule examination must take. Specifically, they must check the satisfiability of the composition rules from each specific set of services. For the examination of these rules, as it has been referred, we can use a set of provers which take a set of knowledge and a formula specified in the first-order logic as input and try to prove if the formula is satisfiable in this state or
not. For example, when we want to prove that the execution of a service enables the execution of another service, we want to prove that the formula $Post_i \land Inv'_i \Rightarrow Pre_j \land Inv_j$ is satisfiable. The knowledge that we have in this state is that the predicates $Post_i$ and $Inv'_i$ are satisfiable and also that the service $S_i$ has been executed correctly ($Pre_i \land Invi \land Post_i \Rightarrow Inv'_i$). Besides this type of rules, the prover will have to check the satisfiability of a set of conditions (precondition and invariants) in accordance to the initial state. This rule takes place in order to determine if a specific service can be executed in the initial state or not. The basic methodologies in which the implementations of a huge set of provers are based are the Resolution and the Davis-Putman-Logemann-Loveland procedure.

In the Resolution method ([JW76]) all the sentences are converted into conjunctive normal form. In order to prove that something is satisfiable or not, the negation of it is added to the set of axioms in the clause form. The resolution method is sound but the resolution refutation is complete. Refutation means proof by contradiction. Given a knowledge base with a collection of true sentences if we want to prove that the sentence $P$ is true, it is assumed that $P$ is false and a contradiction is tried to be arisen. If the theorem can be proven from the initial clauses, the resolution method will produce a contradiction by generating the empty clause. Thus, the rules, which we want to prove, can be ensured by the knowledge base that we have. This methodology is described as sound and complete. Specifically, it always gives you an answer (complete) and also this answer is always correct due to the sound of the algorithm.

The DPLL method does a backtracking search for a model of the formula. It is a complete method, so it returns a satisfying assignment, if one exists. The main idea of the method is to choose an atom from the formula and proceed with two recursive calls. In the first, this atom is obtained while in the second the negation of the atom. The method terminates with the answer “unsatisfiable” if all the branches of the tree concluded in the empty clause, otherwise it returns “satisfiable”. The DPLL is an effective method for SAT problems. Some of the modern SAT solvers that are based on the DPLL are Chaff ([MMM'01]), and e.t.c. The SAT provers return as an answer to an assignment of the predicates that compose the formula, if there is one.

One of the provers that are based on the resolution method is the Otter ([Kal01]) ([McC90]). The Otter (Organized Techniques for Theorem-proving and
Effective Reasoning) is a current automated deduction system that was designed to prove theorems stated in first-order logic with equality. In Otter the inference rules are based on resolution and paramodulation. Also, it uses facilities for term rewriting and ordering and a set of strategies for directing and restricting searches for proofs. It offers a strong soundness guarantee. The inputs of the Otter theorem prover is a list of clauses that are satisfied and the negation of the theorem that we want to prove. It takes the negation of the theorem because it tries to prove the refutation of the theorem negation. Each first-order statement can be converted to clause form by removing universal quantifiers and transforming to disjunction. If after the implication of the inference rules used by the prover, the Otter returns false, it has proven the theorem by the defined clauses.

The EQP (Educational Theorem Prover) ([EQP]) is a variant of Otter. The main difference between these two provers is that EQP is restricted to first-order equational logic. Specifically, all the binary operators are assumed to be associative and commutative.

Another theorem prover which solves problems in first-order logic is the Ordered Semantic Hyperlinking or OSHL ([PZ00]). The OSHL generates models and instances of input clauses contradicting the models and uses them to aid the theorem proving process. The OSHL is sound and complete in first-order logic and applies propositional methods in order to solve the problems that have been defined in the first-order logic. In OSHL there is no backtracking, once a rule is applied, it is never undone.

LeanTaP ([BP95]) is a tableaux theorem prover for first-order logic without identity. The LeanTaP has the best inference rate on simple to moderately complex problems. It is said that the performance is comparable to even much more elaborate systems like Otter.

Setheo (Sequential THEOrem prover) ([BBL’92]) is a high performance automated prover. It uses model elimination respectively to the connection tableau calculus, in order to prove the satisfiability of a sentence. Specifically, it is based on the calculus of so called “connection tableaux”. This calculus can be seen as an efficient integration of the tableau calculus, model elimination and the connection method. Model elimination is a process which tries to prove an unsatisfiable of a formula by eliminating all satisfying assignments. The model elimination can be
implemented with the repeated application of the resolution operation. Model elimination by using only the resolution procedure is sufficient to determine the satisfiability of CNF formula. If the formula is not satisfiable the repeated application of the resolution will result in the production of an empty clause \{\}. The Setheo works differently from other provers that use the resolution procedure. The resolution based provers’ use a breadth first search while Setheo a depth first search controlled by backtracking.

Finally, Gandalf ([Tam97]) is a resolution theorem prover. It supports first-order classical logic with equality, first-order intuitionistic logic with equality, propositional linear logic and a fragment of Martin-Lof type theory. It is characterized for speed and specialises in manipulation of large clauses.

From the representation of the above provers for first-order logic sentences, we can conclude that the theorem Otter is a suitable prover that can be used for the proof of the satisfiability of the specific rules during plan generation. Otter is sound and complete, so it always gives us an answer and it will always be this correct. The provers used for the solution of the SAT problems are not so appropriate for our case. These solvers try to find an assignment for the predicates of the formula, if there is one. However, our intention is to take an answer if a specific formula is satisfiable or not, and not a set of assignments that prove the formula satisfiable.

### 4.4 Conclusions

In this chapter we present how the composition rules can be used in the planners that based on the backward approaches. It is made clear, that their insertion in the plan generation provide the planner with the ability to produce always a plan with a set of services, the execution of which will not produce any conflict and always will provide the desired goal.

The rule definitions with the use of the appropriate rule languages (chapter 2) can be inserted in the Web service specification languages. The augmentation of the specification languages with the rules provide them with the ability to determine the inputs, outputs, preconditions, postconditions and invariants of the complex services.

Vassiliki Alevizou
Chapter 5

Augmenting OWL-S with the Composition Rules defined in SWRL FOL

The rules that have been defined in our model, as it has been presented, are based on the conditions of the Web services that participate in a specific composition. So, the standard that can be selected for it augmentation with these rules, must provide in Web services specification the definition of the appropriate conditions, preconditions, postconditions and invariants. From the standards that can be used for the Web services specification and have been presented in the chapter 2, it is made obvious that the OWL-S is the most suitable. In the OWL-S Web service specification language, a composite process decomposed into a set of subprocesses. The composition is specified using several control constructs: Sequence, Split, Split+Join, Any-Order, Choice, If-Then-Else, Repeat-While and Repeat-Until. However, WSMO at its current status does not provide any means to define the orchestration of the services that participate for the service production. Orchestration describes how the service works from the provider’s perspective. Specifically, how a service makes use of other services or goals for its production.

Although the OWL-S specification language has an expressive power for the definition of the Web services, it has limitations to what can be said about the properties of the ontologies. The OWL language is used for the definition of the ontologies while the OWL-S for Web services specification. In OWL-S is impossible to define relationships between two properties, but this problem is faced with the definition of the rule languages such as SWRL and SWRL FOL. Both of these languages have been presented in chapter 2, analytically.
5.1 **Comparing SWRL and SWRL FOL**

As it is presented in chapter 2, both of them are designed to describe rules and constraints in the framework of the OWL language. They define rules with the following form: “if $A$ and $B$ then $C$”. The meaning of the rule can be read as: whenever the conditions specified in the antecedent hold, then the conditions specified in the consequent must also hold. SWRL extends OWL DL with Horn rule, while SWRL FOL inserts the FOL language in the syntax on the OWL DL. Specifically, the First-Order language inserts in the SWRL the propositional constructs (and, or, neg) and the typed quantifiers (forall, exists). SWRL in its abstract syntax defines the quantifiers “forall” and “exists” but not explicitly. When a rule is defined in the SWRL exists for all the individual objects, while the quantifier “exists” can be defined with the axiom “someValuesfrom”. In addition in the SWRL FOL we can extend the rules in the following form: “If $A$ and $B$ then $C$ or $D$” defining in the head of a rule alternative cases. So, except from conjunctions we can also define disjunctions in the head of the rule. The extension of the SWRL with the first-order language provide us with the ability to define implications inside of other implications. Finally, the SWRL FOL presents the propositional construct “Neg” which can be used for the definition that something does not exist. In the SWRL FOL is allowed on all levels of rulebase elements the use of explicit quantifiers “Forall” and “Exists”.

Considering the above comparison between SWRL and SWRL FOL for defining rules in the OWL-S ontology, it is obvious that the SWRL FOL gives us more abilities in rule definitions than SWRL. Thus, the language that we choose in order to define the composition rules, which must be satisfied in each type of composition, in the OWL-S ontology is the SWRL FOL language. With the use of the SWRL FOL, we have the ability to assert that the relation between two properties may not be existed for a specific set of instances of these two classes. This assertion can be made with the use of the propositional construct “Neg” which introduced in the definition of the SWRL FOL. This ability is not provided by the SWRL language due to the lack of the construct “Neg”. The rule definitions with the SWRL FOL are presented analytically in the Appendix A. Moreover, we presents rules in the SWRL FOL that determine the exact preconditions, postconditions, inputs, outputs and

University of Crete, Computer Science Department
invariants which must have a composite Web service from the corresponding preconditions, postconditions, invariants, inputs, and outputs of the composite Web services.

5.2 *Changes in the OWL-S ontology*

As it has been referred, in the OWL-S are defined three types of processes, atomic, composite and simple. All these classes are defined as subclasses of the class “Process”.

A composite process is composed of a set of subprocesses, the construction of which determined by a specific subclass of the class “ControlConstruct”. Each control construct is associated with a property known as *components* to indicate the nested control constructs from which it is composed. The complete ontology about the composite process is presented below (figure 5-1). The class “Perform” is a kind of control constructs specifying where the client should invoke a process provided by some server. “Perform” class may contain references to atomic or composite processes.

![Figure 5-1: The control constructs that provided in the OWL-S ontology](image)

The OWL-S ontology defines only conditions about the process that must be satisfied in the initial and final state of it, such as the preconditions and the effects, respectively. The preconditions are conditions that must be satisfied in the initial state of the process in order that the execution of the service will be performed
For the definition of the conditions can be used three types of languages such as SWRL, DRS, and KIF. All these conditions are instantiated as instances of the class “Condition” with a property expressionBody that presents the specific statement of the condition. Each condition is presented as an XMLLiteral (e.g. string) in which can be implemented a set of functions (built-ins). However the effects are conditions that satisfied after the performance of a process and result changes in the state of the world. Specifically, the OWL-S does not relate directly a process with the effects. In addition each effect is related with a condition which must be satisfied in order to be applied the effect. In the same way, effects are expressions that can be defined with the use of the languages, SWRL, DRS and KIF. In accordance to the above, for the correlation of each process with a set of conditions different from the preconditions and the effects, which must be satisfied in the initial and final state of the process, we propose the definition of a new property named hasInvariant which will link each process with all the conditions that must be satisfied not only in the initial state but also in the final state of the service.

The class “CompositeProcess” inherits all the properties of the superclass “Process”. Except from these properties, in class “CompositeProcess” have been defined the properties computedInput, computedOutput, computedPrecondition, and computedEffect. These properties contain the inputs, outputs, preconditions and effects of a composite process. Specifically, each of them contains a single expression that characterizes the inputs, outputs, preconditions and effects of a composite process. All of these are named with the prefix “computed” because these properties will be computed automatically by inspecting the makeup of the composite process. For example, a computed precondition is a single expression that characterizes the preconditions required of a composite process, based on the preconditions of its subprocess. Until now, is not used a specific language for the definition of these expressions, so the properties have as range the class “Thing”.

As all these properties contain information about a composite process which produced with the composition of a set of subprocesses, it will be logical if they refer to the same classes that have as ranges the corresponding properties of the class “Process”. For example the property computedInput can have as range the class Input with instances all the inputs of the composite process. Respectively, the property computedPrecondition can refer to the class “Condition”, describing the conditions
under which the execution of the composite process will be sufficient. For the
definition of the invariants of the composed process must be defined a new property
\textit{computedInvariant} which will correlate each composed process with a set of
invariants. The invariants will be determined by the invariants of the subprocesses in
accordance to the type of composition. The values of all these properties that refer to a
specific complex process, are determined at runtime from the composition rules that
we have defined with the use of the SWRL FOL language and must be satisfied in
each of complex iteration.

As the computed properties contain information about the composite
processes, they must have maxcardinality more than one, because a composite process
can have more than one inputs, preconditions e.t.c. With the use of the SWRL FOL,
we can define assertions with which we can relate each instance of a specific property
of a subprocess with the respectively computed property of the composite process.
For example, we can define that the instance ranges of the property \textit{hasInput} of a
process that participate in the sequential composition of a complex service comprise
the inputs of this composite process. All these rules can be specified with the use of
the SWRL FOL language and are presented analytically in the Appendix A.

5.3 \textit{Assumptions in SWRL FOL rules due to the OWL-S ontology}

In the rules that we will define with the use of the SWRL FOL language, we have
considered that each complex process is analyzed in a set of primitive processes, and
each class \textit{"Perform"} refers only to Atomic processes. So, each
\textit{"ControlConstructList"} or \textit{"ControlConstructBag"} that linked with the class
\textit{"ControlConstruct"} with the property \textit{components} is consisted of primitive processes
and not from other type of control constructs (figure 5-1). This assumption is made
because in the SWRL FOL language we cannot refer into predefined rules in the
definition of a new rule. In addition, there is not the notion of regression in SWRL
FOL rules definition.

The control constructs \textit{"Sequence"} and \textit{"Parallel"} are linked with the classes
\textit{"ControlConstructList"} and \textit{"ControlConstructBag"} by the property \textit{components},
respectively. This property has cardinality 1 which means that each instance of the

Vassiliki Alevizou
classes “Sequence” or “Parallel” can be correlated with only one instance of the classes “ControlConstructList” or “ControlConstructBag”. With the composition rule “Post$_i$ \& Inv’$_i$ \Rightarrow Pre$_{i+1}$ \& Inv$_{i+1}$” in the sequential composition, we want to define that between the processes which participate in the sequential composition with the order that determines by the controlconstruct list that the postconditions and the invariants of the previous process determine the preconditions and the invariants of the next process. However, this rule cannot be defined explicitly with the SWRL FOL rule language. With the form of the SWRL FOL rules and the operators that provides, does not give us the ability to take the process that participate in the sequential composition with order. This rule can be defined between all the processes that belong in a “ControlConstructList”. The same is stand for the rules of the parallel composition.

In addition, in the rule definitions as Inv’ we symbolize the invariants that must be satisfied in the new values of the variables that resulted after the service execution. The difference between the Inv and Inv’ is based on the values of the variables, but this difference cannot be defined in the OWL-S ontology and also in the SWRL FOL rules. Finally, we have considered that the expressions and the conditions are specified with the KIF language and the range of the property expressionBody has range String. This assumption provides us to use built-ins for examining the values of the conditions and expressions.

5.4 Rules and Graph instantiation in the OWL-S ontology

In the below sections are presented some examples that pictures the graph instantiation of some services with the use of the OWL-S ontology and also the graph representation of each complex service which produced from the composition of them. In addition, they presented the values of the properties of the complex services such as computedInput, computedPrecondition, based on the values of the subservices that participate in a specific composition. The explicitly values of some of the properties are being determined at runtime by the operation of the composition rules.
5.4.1 Sequential Composition

The OWL-S ontology that represents a composite process from the sequential composition of a set of processes is pictured below. The class “Sequence” which is one of the available subclasses of the class “ControlConstruct” is defined as having a list of component processes that specify the body.

In order to be a sequential composition efficient, the participant services must satisfy the following rule:

\[ \text{Post}_i \land \text{Inv}'_i \Rightarrow \text{Pre}_{i+1} \land \text{Inv}_{i+1} \]

With the above rule, we preclude the case in which the execution of the previous service prohibits the execution of the next service. The properties of a composite process are determined by the properties of the processes, which participate in the composition. For example, we consider that we have two services, a Ticket reservation service (S₁) and a Room reservation service (S₂), respectively. The graph representations of these two services are presented below:
In these graphs are described the two processes, by introducing the inputs, and the outputs of them. In addition are determined the conditions under which the specific process can be executed, specifically, the service \( S_1 \) (ticket_reserve) has as precondition the statement “\( \text{ticket\_avail} \geq \text{ticket\_num} \)” in accordance with, the process will be executed only in the case where the number of the available tickets is bigger than the number of the reservations that a user wants to make. Each service’s effects and outputs is related with a specific condition, the value of which determines the effects and the outputs that will be occurred at the end of the service. If the condition “\( \text{ticket\_avail} \geq \text{ticket\_num} \)” is true, then the effect “\( \text{ticket\_reserve} (\text{ticket\_num}) \& \text{ticket\_avail} = \text{ticket\_avail} - \text{ticket\_num} \)” will take place. The service \( S_1 \) will be in a consistence state if the condition “\( \text{total\_booking} \leq \text{limit\_booking} \)” is being satisfied.

In the sequential composition, the inputs of the composite process are determined by the input of the process that will be executed first. In addition, the outputs of the composite process are the outputs of the last subprocess in the execution row, and also the postconditions of this composite process are the postconditions of the process that is executed last. In OWL-S is not defined a class naming as “Postcondition”. It is defined a class called “Result” which relates each efficient effect and output with a condition. Depending on the conditions values, a specific effect and output will take place. In the following picture we present the schema of the OWL-S ontology about a composite process that produced with the sequential composition of two services and the graph instantiation of a composite service which produced from the sequential composition of the services \( S_1 \) and \( S_2 \). Note that in this figure are determined the inputs, outputs, preconditions, invariants.
and effects of the composite process by the inputs, preconditions and invariants of the service $S_1$ and the outputs and effects of the service $S_2$.

Figure 5-5: The graph representation of the sequential composition of a Ticket reservation and a Room reservation service

5.4.2 Parallel Composition

In parallel composition the rules specification of a complex Web service are defined with the use of the following OWL schema. The control construct “Split” determines that the composite process consists of concurrent execution of a branch of subprocesses.

Figure 5-6: The OWL-S ontology for the Parallel composition
In OWL-S ontology, there is no further specification about waiting and synchronization, so we consider that all the processes begun to executed simultaneously. Thus, in the initial state must be satisfied the preconditions and the invariants of each subprocess.

\[ Pre_1 \land Inv_1 \land Pre_2 \land Inv_2 \land \ldots \land Pre_i \land Inv_i \]

In addition, the set that composed of the invariants of each subprocess must be consistent in order to preclude cases where the execution of one process obligates the execution of another. Thereby the following condition must be satisfied.

\[ Inv_1 \land Inv_2 \land \ldots \land Inv_i \]

When all processes have already been executed, the postconditions and the invariants must be satisfied.

\[ Post_1 \land Inv_1 \land Post_2 \land Inv_2 \land \ldots \land Post_i \land Inv_i \]

The composite process can be described as a black box, which produces a set of information (outputs and effects) considering a set of inputs and constraints. So, the inputs, preconditions and invariants of the composite process are determined by the inputs, preconditions and invariants of all the subprocesses, respectively. Also the outputs and postconditions of the composite process are composed of the outputs and postconditions of the subprocesses.

Supposing that the services S_1 and S_2 make online transactions using a specific credit card. If a user wants to buy a book and a music album, the only thing that must be made, is to be executed these two services which will materialize the transactions. These two services will be able to be executed in parallel because there is not a specific constraint which makes one of these services to wait until the end of the execution of the other. The graph representation of these services are presented in the below figure, in which made clear the inputs, preconditions, invariants and postconditions that the services have.
In accordance with all that mentioned above, the graph that pictures the representation of the composite process which produced from the parallel composition of the above services is presented in the following figure. It shows that the composite process has as input the number of the credit card with which the user will make the transactions. Also, the postconditions that will be produced after its execution, if there will be in stock the appropriate book and cd. The postconditions will be the shipping of these two things and the charge of the credit card with the appropriate amount.
The use of the SWRL FOL does not give as the possibility to determine the values of the properties `computedInput`, `computedOutput`, `computedPrecondition` and `computedEffect` of a composite process from the subservices which participate in parallel composition. The SWRL FOL rules do not provide as the ability to define that something is equal with the conjunction of more than one thing. For example, we cannot say that \( P_{\text{computedInput}} = P_{1\text{input}} \land P_{2\text{input}} \land P_{3\text{input}} \) where \( P_1 \), \( P_2 \), and \( P_3 \) are the services that take part in parallel composition. Also in the syntax of the SWRL FOL does not allow the recursion, so we cannot take one after the other the processes in order to determine the conditions of the composite process.

### 5.4.3 Iterational Composition

The iterate construct makes no assumption about how many iterations are made or when to initiate, terminate, or resume. The initiation, termination or maintenance condition could be specified with a `whileCondition` or an `untilCondition`.
We consider that the condition is implemented with the property \textit{whileCondition}. The property \textit{whileCondition} has as range an instance of the class “Condition” and this condition determines the initiation, execution and termination of the process. So, the rules that must be satisfied in this type of composition are the same with that of the \textit{Repeat–While} control construct, in which we check the value of the condition every time before we start to execute the process. Thus, before the execution of the service we have to check if the following condition is satisfied or not. In the case where the statement is indulged the service will be executed for another more time, else it terminates.

In general, in the iterational composition, before a service starts to be executed, the following condition must be satisfied: $Post_{i-1} \land Inv'_{i-1} \Rightarrow Pre_i \land Inv_i$.

Now, where the Iterate control construct implemented with the use of the Repeat-While control construct, we change this condition into the below form, where $G$ refers to the condition that determines the number of the iteration.

$$Pre_{i-1} \land Inv_{i-1} \land Post_{i-1} \land G \Rightarrow Inv'_{i-1}$$
In other words, the service will be continue to be executed if the condition G and the conditions of the service entail the satisfaction of the invariants of it in the new values of the variables.

Supposing, that we have in our disposal a service for withdrawals from a specific bank account. The preconditions, inputs, invariants, effects and outputs of that service are presented in the below figure. The figure pictures the graph representation of it. As, it is shown, the service in order to start its execution it wants an account number, which must be valid. If the amount is smaller or equal of 50$ and there are enough money in the account, the withdrawal will be completed and the balance will be decreased by 50$.

![Figure 5-12: The graph representation of a Withdrawal service](image)

If we want to withdrawal more than 50$, we must execute the above service more that once time. In the case, where the total amount that we want is 150$, considering the above, the Withdrawal service must be executed 3 times. The composite service, which produced from the iterational execution of the Withdrawal service, is presented below. The composite process has as inputs, preconditions, invariants, effects and outputs the same with the Withdrawal service. The condition \(\text{iteration\_num} < 3\) is that, which will determine the number of service execution.
5.4.4 Repeat-Until Composition

The class “Repeat-Until” specializes the “ControlConstruct” class with the properties untilCondition, which has range the class “Condition”, and the property untilProcess that refers into a control construct. As in the previous cases, also in that case we consider that the control construct is consisted only from an atomic process.

In the ‘Repeat-Until’ composition the service continues to be executed until a specific condition becomes true. For example, if a Web service has the following constraints (Pre, Inv, and Post) it will continue to be executed until the following condition is being satisfied:

Vassiliki Alevizou
\[ \neg G \land Pre' \land Inv' \land Post' \Rightarrow Inv'' (1) \]

In this complex composition, all the constraints of the complex Web service are determined by the constraints of the service, which participate in the composition. That, because is used only one service which is executed in succession until a specific condition became true. Also, the inputs and outputs of the composite process determined by the inputs and the outputs of the process that refers in the control construct “Repeat-Until”.

For the graph representation of the properties of a composite process that produced with the execution of a specific service until a condition become true, we use as an example the service, which enrolls students in a specific course. In the below figure, we can see that the enrollment of a specific student will be made only in the case where this student also attends the course “hy140”. The process will continue to be executed until the condition \( \text{current\_size(course)} > \text{total\_size(course)} \) became true and the constraints of the process that presented in the below figure, satisfy the statement (1).

![Figure 5-15: The graph representation of a service that enrolls students in a course](image)

The composite process \( S_{\text{composite}} \), as we can see in the below figure has as input the number of the students that we want to enroll in a specific course and a list with the name of the students “list\_of(x)”.

University of Crete, Computer Science Department
5.4.5 If-Then-Else Composition

The class “ControlConstruct” specialized also by the control construct “If-Then-Else”. This class consisted of a condition, and two processes. The value of this condition determines which of these two processes will be executed. If the value of the condition is true, the process that refers to the property then is executed, else the process in which is referred the property else. These two properties are presented in the following figure, which represent the ontology for a composite process composed of the complex iteration “If-Then-Else”.

Figure 5-17: The OWL-S ontology for the If-Then-Else composition
The examination of which of these two services will be executed determined by the values of the following conditions.

1. \( G \land \text{Pre}_x \land \text{Inv}_x \land \text{Post}_x \Rightarrow \text{Inv}'_x \)
2. \( \neg G \land \text{Pre}_y \land \text{Inv}_y \land \text{Post}_y \Rightarrow \text{Inv}'_y \)

The values of the properties \text{computedInput}, \text{computedOutput}, \text{computedPrecondition}, \text{computedEffect}, and \text{computedInvariant} of the complex process are determined by the property values \text{hasInput}, \text{hasOutput}, \text{hasPrecondition}, \text{hasEffect}, and \text{hasInvariant} of the subprocess respectively. Depending on the condition value, these properties may have as value either the values of the process, which determined by the property \text{then} or by the property \text{else}. So, for each property, we have to define two alternative rules depending on the condition values (true or false). These rules have been defined with the use of the SWRL FOL language and are presented in the Appendix A.

Supposing, that we have to reserve a hotel room in a city for some days. If the hotel is in the outskirts of the city, we want to rent a car for our transportation, else we can rent a bike. For the accomplishment of this goal we can execute either a service that rent cars or a service that rents bikes. But, which of these two services will execute finally, will be determined by a condition that based on the hotel location. These two services are presented below with all the constraints that have.

Figure 5-18: The graph representation of a service that rent cars
Figure 5-19: The graph representation of a service that rent bikes

The composite service that produced from the composition of these two is presented below. This service will have as inputs, outputs, preconditions, invariants and effects, those that will be produced from the execution of one two services. The selection of the service will be determined by the value of the condition. In the below figure are pictured the conditions of both services, but at runtime only the conditions of one of these services will be taken place.

Figure 5-20: The graph representation of the composite service that produced from the If-Then-Else composition of a Rent_bike service and a Rent_car service
5.5 **Conclusions**

In these sections are compared the rule languages that can be used for the definition of rules in the framework of OWL. They were presented the changes that must be made in the OWL-S ontology for the definition of the rules and also the assumptions, which must be made due to the specification of the SWRL FOL rule language and the OWL-S ontology. Finally, by the representation of a set of examples, is made obvious how the service models of the services that participate in each type of complex composition correlated and produce the specification of the service model of the complex service. The correlation of all the services is determined at runtime by the composition rules. The rules definitions are presented in the Appendix A.
Chapter 6

Conclusion and Future Work

6.1 Conclusion

As the number of deployed Web services increases, the ability to select the most appropriate service for satisfying a user’s needs is an important aspect for the development of Web service standards and applications. Moreover, this need becomes more imperative, if there are no Web services that can satisfy the functionality required by the user, so the composition of the already available services is needed.

In this thesis, we gave an overview of recent progress in Web service standards, specification languages and automatic composition. Current ontological specifications for semantically describing properties of Web services are limited to describing properties, which must be satisfied only in the initial or final state, such as preconditions and postconditions, respectively. We proposed the definition of a new set of properties, known as invariants that must be satisfied not only in the initial state but also in the final. This new type of conditions determines the state of the service. Specifically, their satisfaction ensures the consistent state of a service.

Considering that a Web service can be described by its preconditions, postconditions and invariants, we define a set of composition rules that must be evaluated for the services participating in each composition pattern (sequence, parallel, iterate, repeat-until, if-then-else, and repeat-while). The rules consist of predicates defined on the conditions of the services. By the definition of the rules, compositionality knowledge is provided in the composed services. Thus, the rules enable verification of the composed service and also reasoning about the composed service on the basis of the services, which selected for the composition. Thus, it
becomes possible to determine the effects, which will be produced with the composition of specific services under a specific complex composition pattern.

In addition, we show how a Web service composition problem can be seen either as a Workflow or an AI planning problem. The former is suitable in the cases where a process model is provided. In AI planning approaches the assumption that the services can be described by the inputs, outputs, preconditions and postconditions is made. We presented how the composition rules can be inserted in the backward planning approaches in order to augment them with reasoning about service composition and action interactions. Considering that the goal specified in first-order logic a suitable theorem prover can be used for the examination of rule condition satisfiability.

Finally, we chose OWL-S, as it provides Web service specifications with preconditions and postconditions, for augmenting it with the rules defining in the SWRL FOL language. With this augmentation, we determine the inputs, outputs, preconditions, postconditions and invariants of the composite service based on the respective conditions of the component services.

### 6.2 Future Work

In this thesis we analyzed the composition rules that must be followed by the component Web services in order to produce a complex Web service. Given the preconditions, postconditions and invariants of the available component Web services following the rules, we can determine the effects of the complex Web service. We have considered all along that the compositional operators are deterministic. The study of non-deterministic composition patterns is an interesting potential extension of this work. Specifically, we’re interested in determining the effects of a complex Web service in the case that a non-deterministic composition operator is employed. For example, what will be the effects of a service that is composed of the iteration of a service without knowing the number of iterations? Another case of a nondeterministic operator is the choice between services. In this case, we have to pick one of the available services, but we do not know which one will be selected. Thus, we cannot directly specify the effects of the complex service.
In addition, it will be very useful to devise a prototype implementation, in which the effects of the complex Web services will be determined based on the compositional rules that we have presented. The implementation of such a prototype is a complex task due to the limitations in the area of Web service specifications. For example, in the OWL-S ontology, a set of changes must be made for the definition of the invariant assertions.

Finally, the difficulties that have arisen in the definition of the rules in service specifications are due to the limited syntax of the current rule languages. Thus, many augmentations should be made not only in Web service specifications but also in the languages that used for rule definition in the domain of semantic Web.
Bibliography


Vassiliki Alevizou


[IBM02] IBM, Microsoft, and BEA. *Business process execution language for web services, version 1.0.* 2002.


Vassiliki Alevizou


[OWL-S] *OWL-S: Semantic Markup for Web Service*. Available at [http://www.daml.org/services/owl-s/1.0/owl-s.pdf](http://www.daml.org/services/owl-s/1.0/owl-s.pdf)


[RuleML] *The Rule Markup Initiative*. Available at: [http://www.dfki.uni-kl.de/ruleml/](http://www.dfki.uni-kl.de/ruleml/)


Appendix A

Definition of Composition Rules with SWRL FOL

**Sequential Composition**

The preconditions of the composite process are determined by the preconditions of the service that is executed first.

\[ \text{Post}_{i-1} \land \text{Inv'}_{i-1} \Rightarrow \text{Pre}_i \land \text{Inv}_i \]

```
<ruleml:Assertion>
  <swrlx:ClassAtom/>
</ruleml:ClassAtom>
<ruleml:ForAll>
  <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
  <ruleml:And>
    <swrlx:individualPropertyAtom swrlx:property="composedOf">
      <ruleml:var> S_comp </ruleml:var>
      <ruleml:var> S_sequencecontrolconstruct </ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom swrlx:property="first">
      <ruleml:var> C_constructList </ruleml:var>
      <ruleml:var> P_perform </ruleml:var>
    </swrlx:individualPropertyAtom>
    <swrlx:individualPropertyAtom swrlx:property="process">
      <ruleml:var> P_perform </ruleml:var>
      <ruleml:var> P_process </ruleml:var>
    </swrlx:individualPropertyAtom>
  </ruleml:And>
  <ruleml:ForAll>
    <ruleml:var type="ControlConstructList"> C_constructList </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:property="components">
        <ruleml:var> S_sequencecontrolconstruct </ruleml:var>
        <ruleml:var> C_constructList </ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:property="first">
        <ruleml:var> C_constructList </ruleml:var>
        <ruleml:var> P_perform </ruleml:var>
      </swrlx:individualPropertyAtom>
    </ruleml:And>
  </ruleml:ForAll>
  <ruleml:ForAll>
    <ruleml:var type="Condition"> cond_invariant </ruleml:var>
    <ruleml:var type="Condition"> cond_preconditionNext </ruleml:var>
  </ruleml:ForAll>
</ruleml:ForAll>
```
<ruleml:var type="Condition"> cond_{invariantNext} </ruleml:var>
<ruleml:Implies>
<ruleml:body>
<swrlx:individualPropertyAtom swrlx:property="hasInvariant">
<ruleml:var> P_{process} </ruleml:var>
<ruleml:var> cond_{invariant} </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="expressionBody">
<ruleml:var> cond_{invariant} </ruleml:var>
<ruleml:var> x_{invariant} </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:builtinAtom swrlx:builtin="%swrlb;#compare">
<ruleml:var> x_{invariant} </ruleml:var>
<owlx:DataValue type="xsd:string"> true </owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:property="hasResult">
<ruleml:var> P_{process} </ruleml:var>
<ruleml:var> P_{result} </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="inCondition">
<ruleml:var> P_{result} </ruleml:var>
<ruleml:var> cond_{result} </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="expressionBody">
<ruleml:var> cond_{result} </ruleml:var>
<ruleml:var> x_{result} </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:builtinAtom swrlx:builtin="%swrlb;#compare">
<ruleml:var> x_{result} </ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:property="hasEffect">
<ruleml:var> P_{result} </ruleml:var>
<ruleml:var> Expression </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="rest">
<ruleml:var> C_{constructList} </ruleml:var>
<ruleml:var> C_{constructListNext} </ruleml:var>
</swrlx:individualPropertyAtom>
</ruleml:body>
</ruleml:head>
<swrlx:individualPropertyAtom swrlx:property="first">
<ruleml:var> C_{constructNext} </ruleml:var>
<ruleml:var> P_{performNext} </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="process">
<ruleml:var> P_{performNext} </ruleml:var>
<ruleml:var> P_{processNext} </ruleml:var>
</swrlx:individualPropertyAtom>
The inputs of the composite service are being determined by the inputs of the first service in the row.

Vassiliki Alevizou
The outputs are being determined by the outputs of the last process in the row.
The postconditions of the composite process are being determined by the postconditions of the last process in the row.

Vassiliki Alevizou
<ruleml:var> CconstructList </ruleml:var>
<ruleml:var> Pperform </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="process">
<ruleml:var> Pperform </ruleml:var>
<ruleml:var> Pprocess </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="hasResult">
<ruleml:var> Pprocess </ruleml:var>
<ruleml:var> Presult </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="inCondition">
<ruleml:var> Presult </ruleml:var>
<ruleml:var> condresult </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datatypePropertyAtom swrlx:property="expressionBody">
<ruleml:var> condresult </ruleml:var>
<ruleml:var> xresult </ruleml:var>
</swrlx:datatypePropertyAtom>
<swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
<ruleml:var> xresult </ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string">true</owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:property="hasEffect">
<ruleml:var> Presult </ruleml:var>
<ruleml:var> expreffect </ruleml:var>
</swrlx:individualPropertyAtom>
<ruleml:Neg>
<swrlx:individualPropertyAtom swrlx:property="rest">
<ruleml:var> CconstructList </ruleml:var>
<ruleml:var> CconstructListNext </ruleml:var>
</swrlx:individualPropertyAtom>
</ruleml:Neg>
</ruleml:head>
<swrlx:individualPropertyAtom swrlx:property="computedResult">
<ruleml:var> Scomp </ruleml:var>
<ruleml:var> Presult </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="computedEffect">
<ruleml:var> Presult </ruleml:var>
<ruleml:var> expreffect </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:property="computedinCondition">
<ruleml:var> Presult </ruleml:var>
<ruleml:var> condresult </ruleml:var>
</swrlx:individualPropertyAtom>
</ruleml:head>
</ruleml:Implies>
</ruleml:ForAll>
</ruleml:And>
</ruleml:ForAll>
</ruleml:Assertion>
The preconditions of the composite process are being determined by the preconditions of the first process in the row.

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:property="composedOf">
        <ruleml:var> S_comp </ruleml:var>
        <ruleml:var> S_sequencecontrolconstruct </ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:property="components">
        <ruleml:var> S_sequencecontrolconstruct </ruleml:var>
        <ruleml:var> C_constructList </ruleml:var>
      </swrlx:individualPropertyAtom>
    </ruleml:ForAll>
    <ruleml:ForAll>
      <ruleml:var type="Perform"> P_perform </ruleml:var>
      <ruleml:var type="Process"> P_process </ruleml:var>
      <ruleml:var type="Condition"> cond_pre </ruleml:var>
      <ruleml:Implies>
        <ruleml:body>
          <swrlx:individualPropertyAtom swrlx:property="first">
            <ruleml:var> C_constructList </ruleml:var>
            <ruleml:var> P_perform </ruleml:var>
          </swrlx:individualPropertyAtom>
          <swrlx:individualPropertyAtom swrlx:property="process">
            <ruleml:var> P_perform </ruleml:var>
            <ruleml:var> P_process </ruleml:var>
          </swrlx:individualPropertyAtom>
          <swrlx:individualPropertyAtom swrlx:property="hasPrecondition">
            <ruleml:var> P_process </ruleml:var>
            <ruleml:var> cond_pre </ruleml:var>
          </swrlx:individualPropertyAtom>
        </ruleml:body>
        <ruleml:head>
          <swrlx:individualPropertyAtom swrlx:property="hasPrecondition">
            <ruleml:var> S_comp </ruleml:var>
            <ruleml:var> cond_pre </ruleml:var>
          </swrlx:individualPropertyAtom>
        </ruleml:head>
        <ruleml:Implies>
          <ruleml:ForAll>
            <ruleml:And>
              <ruleml:ForAll>
                </ruleml:ForAll>
              </ruleml:And>
            </ruleml:ForAll>
            </ruleml:Assertion>
```
Parallel Composition

In parallel composition, the invariants of the subprocess must be consistent. So, the following condition must be satisfied: $\text{Inv}_1 \land \text{Inv}_2 \land \ldots \land \text{Inv}_i$. This rule specified in the SWRL FOL in the following table.

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="composedOf">
        <ruleml:var> S_comp </ruleml:var>
        <ruleml:var> SSplitcontrolconstruct </ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:Property="components">
        <ruleml:var> SSplitcontrolconstruct </ruleml:var>
        <ruleml:var> CconstructBag </ruleml:var>
      </swrlx:individualPropertyAtom>
    </ruleml:ForAll>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="first">
        <ruleml:var> CconstructBag </ruleml:var>
        <ruleml:var> Pperform </ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:Property="process">
        <ruleml:var> Pperform </ruleml:var>
        <ruleml:var> Pprocess </ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:Property="hasInvariant">
        <ruleml:var> Pprocess </ruleml:var>
        <ruleml:var> condinv </ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
        <ruleml:var> condinv </ruleml:var>
        <ruleml:var> xinv </ruleml:var>
      </swrlx:datatypePropertyAtom>
      <swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
        <ruleml:var> xinv </ruleml:var>
        <owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
      </swrlx:builtinAtom>
    </ruleml:And>
  </ruleml:ForAll>
</ruleml:And>
</ruleml:Assertion>
```
The services in parallel composition executed independently, so, in the initial state must be satisfied the preconditions and the invariants of each service.

\[
\begin{align*}
\text{\textbf{CompositeProcess}} \Rightarrow \text{S}\text{\textsubscript{comp}} \\
\text{\textbf{composedOf}} \Rightarrow \text{S}\text{\textsubscript{comp}} \land \text{S}\text{\textsubscript{Splitcontrolconstruct}} \\
\text{\textbf{components}} \Rightarrow \text{S}\text{\textsubscript{Splitcontrolconstruct}} \land \text{C}\text{\textsubscript{constructBag}} \\
\text{\textbf{ControlConstructBag}} \Rightarrow \text{C}\text{\textsubscript{construct}} \\
\text{\textbf{Process}} \Rightarrow \text{P}\text{\textsubscript{process}} \\
\text{\textbf{Condition}} \Rightarrow \text{cond}\text{\textsubscript{pre}} \land \text{cond}\text{\textsubscript{inv}} \\
\text{\textbf{rest}} \Rightarrow \text{C}\text{\textsubscript{constructBagNext}} \land \text{C}\text{\textsubscript{construct}} \\
\text{\textbf{first}} \Rightarrow \text{P}\text{\textsubscript{perform}} \land \text{C}\text{\textsubscript{construct}} \\
\text{\textbf{process}} \Rightarrow \text{P}\text{\textsubscript{process}} \land \text{P}\text{\textsubscript{perform}} \\
\text{\textbf{hasPrecondition}} \Rightarrow \text{P}\text{\textsubscript{process}} \land \text{cond}\text{\textsubscript{pre}} \\
\text{\textbf{expressionBody}} \Rightarrow \text{P}\text{\textsubscript{process}} \land \text{cond}\text{\textsubscript{pre}} \\
\text{\textbf{compare}} \Rightarrow \text{P}\text{\textsubscript{process}} \land \text{cond}\text{\textsubscript{pre}} \\
\text{\textbf{string}} \Rightarrow \text{true} \\
\text{\textbf{hasInvariant}} \Rightarrow 
\end{align*}
\]
Finally, when all processes have already been executed, the postconditions and the invariants must be satisfied.
Iterational Composition

The control construct “Iterate” has not been implemented in the OWL-S (1.1). For the
definition of the rules can be used the control construct “Repeat-While”. In this
construct, the following rule must be satisfied.

\[ G \land Pre_i \land Inv_i \land Post_i \implies Inv_i' \]

Vassiliki Alevizou
\[
\text{\texttt{ruleml:var}} \ S_{\text{comp}} \ \text{\texttt{ruleml:var}} \\
\text{\texttt{ruleml:var}} \ S_{\text{Repeat\_Whilecontrolconstruct}} \ \text{\texttt{ruleml:var}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]

\[
\text{\texttt{ruleml:ForAll}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Condition}} \ \text{\texttt{condWhile}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Condition}} \ \text{\texttt{condpre}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Condition}} \ \text{\texttt{condinv}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Expression}} \ \text{\texttt{exprf}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Perform}} \ \text{\texttt{Pperform}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Condition}} \ \text{\texttt{condin}} \\
\text{\texttt{ruleml:Implies}} \\
\text{\texttt{swrlx:individualPropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{whileCondition}} \\
\text{\texttt{ruleml:var}} \ S_{\text{Repeat\_Whilecontrolconstruct}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{condwhile}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]

\[
\text{\texttt{swrlx:datatypePropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{expressionBody}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{condwhile}} \\
\text{\texttt{ruleml:var}} \ x_{\text{While}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]

\[
\text{\texttt{swrlx:builtinAtom}} \ \text{\texttt{swrlx:builtin}} = \ &swrlb;\#compare \\
\text{\texttt{ruleml:var}} \ x_{\text{While}} \\
\text{\texttt{owlx:DataValue}} \ \text{\texttt{owlx:datatype}} = \ \text{\texttt{xsd:string}} \ \text{\texttt{true}} \\
\text{\texttt{swrlx:datatypePropertyAtom}}
\]

\[
\text{\texttt{swrlx:individualPropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{whileProcess}} \\
\text{\texttt{ruleml:var}} \ S_{\text{Repeat\_Whilecontrolconstruct}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Pperform}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]

\[
\text{\texttt{swrlx:individualPropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{process}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Pperform}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Pprocess}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]

\[
\text{\texttt{swrlx:individualPropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{hasPrecondition}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Pprocess}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{condpre}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]

\[
\text{\texttt{swrlx:datatypePropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{expressionBody}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{condpre}} \\
\text{\texttt{ruleml:var}} \ x_{\text{pre}} \\
\text{\texttt{swrlx:datatypePropertyAtom}}
\]

\[
\text{\texttt{swrlx:builtinAtom}} \ \text{\texttt{swrlx:builtin}} = \ &swrlb;\#compare \\
\text{\texttt{ruleml:var}} \ x_{\text{pre}} \\
\text{\texttt{owlx:DataValue}} \ \text{\texttt{owlx:datatype}} = \ \text{\texttt{xsd:string}} \ \text{\texttt{true}} \\
\text{\texttt{swrlx: builtinAtom}}
\]

\[
\text{\texttt{swrlx:individualPropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{hasInvariant}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{Pprocess}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{condinv}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]

\[
\text{\texttt{swrlx:datatypePropertyAtom}} \ \text{\texttt{swrlx:Property}} = \ \text{\texttt{expressionBody}} \\
\text{\texttt{ruleml:var}} \ \text{\texttt{condpre}} \\
\text{\texttt{swrlx:individualPropertyAtom}}
\]
The inputs of the process that produced with the control construct “Repeat-While” will have the inputs, outputs, preconditions, invariants and postconditions of the
process that will be participate in the composition. The definitions of these rules in the SWRL FOL are presented in the following tables, respectively.

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="Composite Process"> Scomp </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="composedOf">
        <ruleml:var> Scomp </ruleml:var>
      </swrlx:individualPropertyAtom>
      <ruleml:ForAll>
        <ruleml:var type="Input"> Pinput </ruleml:var>
        <ruleml:Implies>
          <ruleml:body>
            <swrlx:individualPropertyAtom swrlx:Property="whileProcess">
              <ruleml:var> SRepeat_Whilelcontrolconstruct </ruleml:var>
              <ruleml:var> Pperform </ruleml:var>
            </swrlx:individualPropertyAtom>
            <swrlx:individualPropertyAtom swrlx:Property="process">
              <ruleml:var> Pperform </ruleml:var>
              <ruleml:var> Pprocess </ruleml:var>
            </swrlx:individualPropertyAtom>
            <swrlx:individualPropertyAtom swrlx:Property="hasInput">
              <ruleml:var> Pprocess </ruleml:var>
              <ruleml:var> Pinput </ruleml:var>
            </swrlx:individualPropertyAtom>
          </ruleml:body>
          <ruleml:head>
            <swrlx:individualPropertyAtom swrlx:Property="computedInput">
              <ruleml:var> Scomp </ruleml:var>
              <ruleml:var> Pinput </ruleml:var>
            </swrlx:individualPropertyAtom>
          </ruleml:head>
          <ruleml:Implies>
            <ruleml:ForAll>
              <ruleml:And>
                <ruleml:ForAll>
                  <ruleml:var type="Composite Process"> Scomp </ruleml:var>
                  <swrlx:individualPropertyAtom swrlx:Property="composedOf">
                    <ruleml:var> Scomp </ruleml:var>
                    <ruleml:var> SRepeat_Whilelcontrolconstruct </ruleml:var>
                  </swrlx:individualPropertyAtom>
                </ruleml:ForAll>
              </ruleml:And>
            </ruleml:ForAll>
          </ruleml:Implies>
        </ruleml:Implies>
      </ruleml:ForAll>
    </ruleml:And>
  </ruleml:ForAll>
</ruleml:Assertion>
```

University of Crete, Computer Science Department
\[
\begin{align*}
\text{<ruleml:ForAll>}
\text{<ruleml:var type="Output"> } P_{\text{output}} \text{ </ruleml:var>}
\text{<ruleml:Implies>}
\text{<ruleml:body>}
\text{<swrlx:individualPropertyAtom swrlx:Property="whileProcess">}
\text{<ruleml:var> } S_{\text{Repeat_Whilecontrolconstruct}} \text{ </ruleml:var>}
\text{<ruleml:var> } P_{\text{perform}} \text{ </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{</ruleml:body>}
\text{<swrlx:individualPropertyAtom swrlx:Property="process">}
\text{<ruleml:var> } S_{\text{Repeat_Whilelcontrolconstruct}} \text{ </ruleml:var>}
\text{<ruleml:var> } P_{\text{process}} \text{ </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{</ruleml:Implies>}
\text{</ruleml:ForAll>}
\text{</ruleml:And>}
\text{</ruleml:ForAll>}
\text{</ruleml:Assertion>}
\end{align*}
\]
\[
\begin{align*}
\text{<ruleml:Assertion>}
\text{<ruleml:ForAll>}
\text{<ruleml:var type="CompositeProcess"> } S_{\text{comp}} \text{ </ruleml:var>}
\text{<ruleml:And>}
\text{<swrlx:individualPropertyAtom swrlx:Property="composedOf">}
\text{<ruleml:var> } S_{\text{comp}} \text{ </ruleml:var>}
\text{<ruleml:var> } S_{\text{Repeat_Whilelcontrolconstruct}} \text{ </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{</ruleml:And>}
\text{<ruleml:ForAll>}
\text{<ruleml:var type="Condition"> } \text{cond}_{\text{pre}} \text{ </ruleml:var>}
\text{<ruleml:Implies>}
\text{<ruleml:body>}
\text{<swrlx:individualPropertyAtom swrlx:Property="whileProcess">}
\text{<ruleml:var> } S_{\text{Repeat_Whilecontrolconstruct}} \text{ </ruleml:var>}
\text{<ruleml:var> } P_{\text{perform}} \text{ </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{<swrlx:individualPropertyAtom swrlx:Property="process">}
\text{<ruleml:var> } P_{\text{perform}} \text{ </ruleml:var>}
\text{<ruleml:var> } P_{\text{process}} \text{ </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{</ruleml:Implies>}
\text{</ruleml:ForAll>}
\text{</ruleml:Assertion>}
\end{align*}
\]
\begin{verbatim}
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="composedOf">
        <ruleml:var> S_comp </ruleml:var>
        <ruleml:var> S_Repeat_Whilecontrolconstruct </ruleml:var>
      </swrlx:individualPropertyAtom>
      <ruleml:ForAll>
        <ruleml:var type="Condition"> cond_inv </ruleml:var>
        <ruleml:Implies>
          <ruleml:body>
            <swrlx:individualPropertyAtom swrlx:Property="whileProcess">
              <ruleml:var> S_Repeat_Whilecontrolconstruct </ruleml:var>
              <ruleml:var> P_perform </ruleml:var>
            </swrlx:individualPropertyAtom>
            <swrlx:individualPropertyAtom swrlx:Property="process">
              <ruleml:var> P_perform </ruleml:var>
              <ruleml:var> P_process </ruleml:var>
            </swrlx:individualPropertyAtom>
            <swrlx:individualPropertyAtom swrlx:Property="hasInvariant">
              <ruleml:var> P_process </ruleml:var>
              <ruleml:var> cond_inv </ruleml:var>
            </swrlx:individualPropertyAtom>
          </ruleml:body>
          <ruleml:head>
            <swrlx:individualPropertyAtom swrlx:Property="computedInvariant">
              <ruleml:var> S_comp </ruleml:var>
              <ruleml:var> cond_inv </ruleml:var>
            </swrlx:individualPropertyAtom>
          </ruleml:head>
        </ruleml:Implies>
      </ruleml:ForAll>
    </ruleml:And>
  </ruleml:ForAll>
</ruleml:Assertion>
\end{verbatim}
Repeat-Until Composition

The symbols Pre’, Inv’, Post’, and Inv” present the constraints that must be satisfied in each state of the Web service execution.

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="composedOf">
        <ruleml:var> S_comp </ruleml:var>
        <ruleml:var> S_{Repeat\_Untilcontrolconstruct} </ruleml:var>
      </swrlx:individualPropertyAtom>
      <ruleml:ForAll>
        <ruleml:var type="Condition"> condRepeat </ruleml:var>
        <ruleml:var type="Condition"> condpre </ruleml:var>
        <ruleml:var type="Condition"> condinv </ruleml:var>
        <ruleml:var type="Expression"> expreff </ruleml:var>
        <ruleml:var type="Perform"> P_{perform} </ruleml:var>
        <ruleml:var type="Condition"> condin </ruleml:var>
      </ruleml:ForAll>
      <ruleml:Implies>
        <ruleml:body>
          <swrlx:individualPropertyAtom swrlx:Property="untilCondition">
            <ruleml:var> S_{Repeat\_Untilcontrolconstruct} </ruleml:var>
          </swrlx:individualPropertyAtom>
          <ruleml:var> condRepeat </ruleml:var>
          <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
            <ruleml:var> condRepeat </ruleml:var>
            <ruleml:var> xRepeat </ruleml:var>
            <swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
              <ruleml:var> xRepeat </ruleml:var>
              <owlx:DataValue owlx:datatype="xsd:string"> false </owlx:DataValue>
            </swrlx:builtinAtom>
          </swrlx:datatypePropertyAtom>
          <swrlx:individualPropertyAtom swrlx:Property="untilProcess">
            <ruleml:var> S_{Repeat\_Untilcontrolconstruct} </ruleml:var>
          </swrlx:individualPropertyAtom>
          <ruleml:var> P_{perform} </ruleml:var>
        </ruleml:body>
      </ruleml:Implies>
    </ruleml:ForAll>
  </ruleml:Assertion>
```
<ruleml:var> P_process </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasPrecondition">
<ruleml:var> P_process </ruleml:var>
<ruleml:var> cond_pre </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
<ruleml:var> cond_pre </ruleml:var>
<ruleml:var> x_pre </ruleml:var>
</swrlx:datatypePropertyAtom>
<swrlx:builtinAtom swrlx:builtin="#swrlb;#compare">
<ruleml:var> x_pre </ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasInvariant">
<ruleml:var> P_process </ruleml:var>
<ruleml:var> cond_inv </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
<ruleml:var> cond_inv </ruleml:var>
<ruleml:var> xin </ruleml:var>
</swrlx:datatypePropertyAtom>
<swrlx:builtinAtom swrlx:builtin="#swrlb;#compare">
<ruleml:var> xin </ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasResult">
<ruleml:var> P_process </ruleml:var>
<ruleml:var> P_result </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:Property="inCondition">
<ruleml:var> P_result </ruleml:var>
<ruleml:var> cond_in </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
<ruleml:var> cond_in </ruleml:var>
<ruleml:var> xin </ruleml:var>
</swrlx:datatypePropertyAtom>
<swrlx:builtinAtom swrlx:builtin="#swrlb;#compare">
<ruleml:var> xin </ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasEffect">
<ruleml:var> P_process </ruleml:var>
<ruleml:var> P_result </ruleml:var>
<ruleml:var> expr_eff </ruleml:var>
</swrlx:individualPropertyAtom>
</ruleml:body>
</ruleml:head>
<swrlx:individualPropertyAtom swrlx:Property="hasInvariant">
<ruleml:var> P_process </ruleml:var>

The inputs of the composite process are being determined by the inputs of the process.

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="Composite Process"> S_comp </ruleml:var>
  </ruleml:ForAll>
  <ruleml:And>
    <swrlx:individualPropertyAtom swrlx:Property="composedOf">
      <ruleml:var> S_comp </ruleml:var>
      <ruleml:var> S_Repea_Untilcontrolconstruct </ruleml:var>
    </swrlx:individualPropertyAtom>
    <ruleml:ForAll>
      <ruleml:var type="Input"> P_input </ruleml:var>
    </ruleml:ForAll>
    <ruleml:Implies>
      <ruleml:body>
        <swrlx:individualPropertyAtom swrlx:Property="untilProcess">
          <ruleml:var> S_Repea_Untilcontrolconstruct </ruleml:var>
          <ruleml:var> P_perform </ruleml:var>
        </swrlx:individualPropertyAtom>
        <swrlx:individualPropertyAtom swrlx:Property="process">
          <ruleml:var> P_perform </ruleml:var>
          <ruleml:var> P_process </ruleml:var>
        </swrlx:individualPropertyAtom>
        <swrlx:individualPropertyAtom swrlx:Property="hasInput">
          <ruleml:var> P_process </ruleml:var>
          <ruleml:var> P_input </ruleml:var>
        </swrlx:individualPropertyAtom>
      </ruleml:body>
    </ruleml:Implies>
  </ruleml:And>
</ruleml:Assertion>
```
The outputs of the composite process are being determined by the outputs of the composite process.

```
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="Composite Process"> S_{comp} </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="composedOf">
        <ruleml:var> S_{comp} </ruleml:var>
        <ruleml:var> S_{Repeat_Untilcontrolconstruct} </ruleml:var>
      </swrlx:individualPropertyAtom>
      <ruleml:ForAll>
        <ruleml:var type="Output"> P_{output} </ruleml:var>
        <ruleml:Implies>
          <ruleml:body>
            <swrlx:individualPropertyAtom swrlx:Property="untilProcess">
              <ruleml:var> S_{Repeat_Untilcontrolconstruct} </ruleml:var>
              <ruleml:var> P_{perform} </ruleml:var>
            </swrlx:individualPropertyAtom>
            <swrlx:individualPropertyAtom swrlx:Property="process">
              <ruleml:var> P_{perform} </ruleml:var>
              <ruleml:var> P_{process} </ruleml:var>
            </swrlx:individualPropertyAtom>
            <swrlx:individualPropertyAtom swrlx:Property="hasOutput">
              <ruleml:var> P_{process} </ruleml:var>
              <ruleml:var> P_{output} </ruleml:var>
            </swrlx:individualPropertyAtom>
          </ruleml:body>
          <ruleml:head>
            <swrlx:individualPropertyAtom swrlx:Property="computedOutput">
              <ruleml:var> S_{comp} </ruleml:var>
              <ruleml:var> P_{output} </ruleml:var>
            </swrlx:individualPropertyAtom>
          </ruleml:head>
        </ruleml:Implies>
      </ruleml:ForAll>
    </ruleml:And>
  </ruleml:ForAll>
</ruleml:Assertion>
```
The preconditions of the composite process are being determined by the preconditions of the process.

\[
\begin{align*}
\text{<ruleml:Assertion>}
\text{<ruleml:ForAll>}
\text{<ruleml:var type="CompositeProcess"> S_{comp} </ruleml:var>}
\text{<ruleml:And>}
\text{<swrlx:individualPropertyAtom swrlx:Property="composedOf">}
\text{<ruleml:var> S_{comp} </ruleml:var>}
\text{<ruleml:var> S_{Repeat\_Untilcontrolconstruct} </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{<ruleml:ForAll>}
\text{<ruleml:var type="Condition"> \textit{cond}_{pre} </ruleml:var>}
\text{<ruleml:Implies>}
\text{<ruleml:body>}
\text{<swrlx:individualPropertyAtom swrlx:Property="untilProcess">}
\text{<ruleml:var> S_{Repeat\_Untilcontrolconstruct} </ruleml:var>}
\text{<ruleml:var> \textit{P}_{perform} </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{<swrlx:individualPropertyAtom swrlx:Property="process">}
\text{<ruleml:var> \textit{P}_{perform} </ruleml:var>}
\text{<ruleml:var> \textit{P}_{process} </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{<swrlx:individualPropertyAtom swrlx:Property="hasPrecondition">}
\text{<ruleml:var> \textit{P}_{process} </ruleml:var>}
\text{<ruleml:var> \textit{cond}_{pre} </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{</ruleml:body>}
\text{<ruleml:head>}
\text{<swrlx:individualPropertyAtom swrlx:Property="computedPrecondition">}
\text{<ruleml:var> \textit{P}_{process} </ruleml:var>}
\text{<ruleml:var> \textit{cond}_{pre} </ruleml:var>}
\text{</swrlx:individualPropertyAtom>}
\text{</ruleml:head>}
\text{<ruleml:Implies>}
\text{</ruleml:ForAll>}
\text{<ruleml:And>}
\text{<ruleml:ForAll>}
\text{</ruleml:Assertion>}
\end{align*}
\]

In addition the invariants of the complex process will be determined by the invariants of the process that will be executed more than one if the condition is being satisfied.

\[
\begin{align*}
\text{<ruleml:Assertion>}
\text{<ruleml:ForAll>}
\text{<ruleml:var type="CompositeProcess"> S_{comp} </ruleml:var>}
\end{align*}
\]
Finally, the postconditions of the complex process that composed with the control construct “Repeat-Until” will be the postconditions of the process that participate in it.
If-Then-Else Composition

In the If-Then-Else composition, the process which will finally be executed depending on the value of the condition G. So, before the execution of one of these services, the following condition must be checked:

\[ G \land Pre_2 \land Inv_2 \land Post_2 \Rightarrow Inv_2 '\]
<ruleml:var type="Composite Process"> S_{comp} </ruleml:var>
<ruleml:And>
  <swrlx:individualPropertyAtom swrlx:Property="composedOf">
    <ruleml:var> S_{comp} </ruleml:var>
    <ruleml:var> S_{if-then-elsecontrolconstruct} </ruleml:var>
  </swrlx:individualPropertyAtom>
<ruleml:ForAll>
  <ruleml:var type="Condition"> cond_{pre} </ruleml:var>
  <ruleml:var type="Condition"> cond_{inv} </ruleml:var>
  <ruleml:var type="Condition"> cond_{result} </ruleml:var>
  <ruleml:var type="Expression"> exp_{effect} </ruleml:var>
  <ruleml:var type="Result"> P_{Result} </ruleml:var>
<ruleml:And>
  <ruleml:Implies>
    <ruleml:body>
      <swrlx:individualPropertyAtom swrlx:Property="ifCondition">
        <ruleml:var> S_{if-then-elsecontrolconstruct} </ruleml:var>
        <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
          <ruleml:var> cond_{if-then-else} </ruleml:var>
          <owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
        </swrlx:datatypePropertyAtom>
      </swrlx:individualPropertyAtom>
      <swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
        <ruleml:var> x_{if-then-else} </ruleml:var>
      </swrlx:builtinAtom>
      <swrlx:individualPropertyAtom swrlx:Property="then">
        <ruleml:var> P_{perform} </ruleml:var>
        <swrlx:individualPropertyAtom swrlx:Property="process">
          <ruleml:var> P_{process} </ruleml:var>
        </swrlx:individualPropertyAtom>
        <swrlx:individualPropertyAtom swrlx:Property="hasPrecondition">
          <ruleml:var> cond_{pre} </ruleml:var>
        </swrlx:individualPropertyAtom>
        <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
          <ruleml:var> cond_{pre} </ruleml:var>
          <owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
        </swrlx:datatypePropertyAtom>
      </swrlx:individualPropertyAtom>
      <swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
        <ruleml:var> x_{pre} </ruleml:var>
      </swrlx:builtinAtom>
      <swrlx:individualPropertyAtom swrlx:Property="hasInvariant">
        <ruleml:var> P_{process} </ruleml:var>
        <ruleml:var> cond_{inv} </ruleml:var>
      </swrlx:individualPropertyAtom>
    </ruleml:body>
  </ruleml:Implies>
\[
\begin{align*}
&\text{\langle/swrlx:individualPropertyAtom}\rangle \\
&\text{\langle/swrlx:datatypePropertyAtom swrlx:Property="expressionBody"\rangle} \\
&\text{\langle/ruleml:var\rangle condinv \langle/ruleml:var\rangle} \\
&\text{\langle/ruleml:var\rangle xinv \langle/ruleml:var\rangle} \\
&\text{\langle/swrlx:datatypePropertyAtom\rangle} \\
&\text{\langle/swrlx:builtinAtom swrlx:Property="swrbl:#compare"\rangle} \\
&\text{\langle/ruleml:var\rangle xinv \langle/ruleml:var\rangle} \\
&\text{\langle/owlx:DataValue owlx:datatype="xsd:string"\rangle true \langle/owlx:DataValue\rangle} \\
&\text{\langle/swrlx:builtinAtom\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom swrlx:Property="hasResult"\rangle} \\
&\text{\langle/ruleml:var\rangle P_{process} \langle/ruleml:var\rangle} \\
&\text{\langle/ruleml:var\rangle P_{Result} \langle/ruleml:var\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom swrlx:Property="inCondition"\rangle} \\
&\text{\langle/ruleml:var\rangle P_{Result} \langle/ruleml:var\rangle} \\
&\text{\langle/ruleml:var\rangle cond_{result} \langle/ruleml:var\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom\rangle} \\
&\text{\langle/swrlx:datatypePropertyAtom swrlx:Property="expressionBody"\rangle} \\
&\text{\langle/ruleml:var\rangle cond_{result} \langle/ruleml:var\rangle} \\
&\text{\langle/ruleml:var\rangle x_{result} \langle/ruleml:var\rangle} \\
&\text{\langle/swrlx:builtinAtom swrlx:builtin="&swrbl:#compare"\rangle} \\
&\text{\langle/ruleml:var\rangle x_{result} \langle/ruleml:var\rangle} \\
&\text{\langle/owlx:DataValue owlx:datatype="xsd:string"\rangle true \langle/owlx:DataValue\rangle} \\
&\text{\langle/swrlx:datatypePropertyAtom\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom swrlx:Property="hasEffect"\rangle} \\
&\text{\langle/ruleml:var\rangle P_{Result} \langle/ruleml:var\rangle} \\
&\text{\langle/ruleml:var\rangle exp_{effect} \langle/ruleml:var\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom\rangle} \\
&\text{\langle/ruleml:body\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom swrlx:Property="hasInvariant"\rangle} \\
&\text{\langle/ruleml:var\rangle P_{process} \langle/ruleml:var\rangle} \\
&\text{\langle/ruleml:var\rangle cond_{inv} \langle/ruleml:var\rangle} \\
&\text{\langle/swrlx:individualPropertyAtom\rangle} \\
&\text{\langle/swrlx:datatypePropertyAtom swrlx:Property="expressionBody"\rangle} \\
&\text{\langle/ruleml:var\rangle cond_{inv} \langle/ruleml:var\rangle} \\
&\text{\langle/ruleml:var\rangle x_{inv} \langle/ruleml:var\rangle} \\
&\text{\langle/swrlx:datatypePropertyAtom\rangle} \\
&\text{\langle/swrlx:builtinAtom swrlx:builtin="&swrbl:compare"\rangle} \\
&\text{\langle/ruleml:var\rangle x_{inv} \langle/ruleml:var\rangle} \\
&\text{\langle/owlx:DataValue owlx:datatype="xsd:string"\rangle true \langle/owlx:DataValue\rangle} \\
&\text{\langle/swrlx:builtinAtom\rangle} \\
&\text{\langle/ruleml:head\rangle} \\
&\text{\langle/swrlx:DataValue\rangle} \\
&\text{\langle/swrlx:builtin\rangle} \\
&\text{\langle/ruleml:head\rangle} \\
&\text{\langle/ruleml:Implies\rangle} \\
&\text{\langle/ruleml:And\rangle} \\
&\text{\langle/ruleml:ForAll\rangle} \\
&\text{\langle/ruleml:And\rangle} \\
&\text{\langle/ruleml:ForAll\rangle} \\
\end{align*}
\]
\[-G \land Pre_3 \land Inv_3 \land Post_3 \Rightarrow Inv'_3\]

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="Composite Process"> S_{comp} </ruleml:var>
  </ruleml:ForAll>
  <ruleml:And>
    <ruleml:individualPropertyAtom swrlx:Property="composedOf">
      <ruleml:var> S_{comp} </ruleml:var>
    </ruleml:individualPropertyAtom>
    <ruleml:individualPropertyAtom swrlx:Property="ifCondition">
      <ruleml:var> S_{if-then-elsecontrolconstruct} </ruleml:var>
    </ruleml:individualPropertyAtom>
    <ruleml:And>
      <ruleml:Implies>
        <ruleml:body>
          <ruleml:individualPropertyAtom swrlx:Property="ifCondition">
            <ruleml:var> S_{if-then-elsecontrolconstruct} </ruleml:var>
          </ruleml:individualPropertyAtom>
          <ruleml:individualPropertyAtom swrlx:Property="expressionBody">
            <ruleml:var> x_{if-then-else} </ruleml:var>
          </ruleml:individualPropertyAtom>
          <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
            <ruleml:var> x_{if-then-else} </ruleml:var>
          </swrlx:datatypePropertyAtom>
          <swrlx:builtinAtom swrlx:builtin="\&swrlb;#compare">
            <ruleml:var> x_{if-then-else} </ruleml:var>
          </swrlx:builtinAtom>
          <swrlx:individualPropertyAtom swrlx:Property="else">
            <ruleml:var> S_{if-then-elsecontrolconstruct} </ruleml:var>
          </swrlx:individualPropertyAtom>
          <swrlx:individualPropertyAtom swrlx:Property="process">
            <ruleml:var> P_{perform} </ruleml:var>
          </swrlx:individualPropertyAtom>
          <swrlx:individualPropertyAtom swrlx:Property="hasPrecondition">
            <ruleml:var> P_{process} </ruleml:var>
          </swrlx:individualPropertyAtom>
          <swrlx:individualPropertyAtom swrlx:Property="expressionBody">
            <ruleml:var> x_{pre} </ruleml:var>
          </swrlx:individualPropertyAtom>
          <swrlx:builtinAtom swrlx:builtin="\&swrlb;#compare">
            <ruleml:var> x_{pre} </ruleml:var>
          </swrlx:builtinAtom>
        </ruleml:body>
        <ruleml:var type="Result"> P_{Result} </ruleml:var>
      </ruleml:Implies>
      <ruleml:And>
        <ruleml:var type="Condition"> cond_{pre} </ruleml:var>
        <ruleml:var type="Condition"> cond_{inv} </ruleml:var>
        <ruleml:var type="Expression"> exp_{effect} </ruleml:var>
      </ruleml:And>
    </ruleml:Implies>
  </ruleml:And>
</ruleml:Assertion>
```
<owlx:DataValue owlx:datatype="xsd:string">true</owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasInvariant">
<ruleml:var>\text{Pprocess}</ruleml:var>
<ruleml:var>\text{cond}_{\text{inv}}</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
<ruleml:var>\text{cond}_{\text{inv}}</ruleml:var>
<ruleml:var>\text{x}_{\text{inv}}</ruleml:var>
</swrlx:datatypePropertyAtom>
<swrlx:builtinAtom swrlx:builtin="&\text{swrlb;#compare}">
<ruleml:var>\text{x}_{\text{inv}}</ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string">true</owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasResult">
<ruleml:var>\text{Pprocess}</ruleml:var>
<ruleml:var>\text{PResult}</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datatypePropertyAtom swrlx:Property="inCondition">
<ruleml:var>\text{PResult}</ruleml:var>
<ruleml:var>\text{cond}_{\text{result}}</ruleml:var>
</swrlx:datatypePropertyAtom>
<swrlx:builtinAtom swrlx:builtin="&\text{swrlb;#compare}">
<ruleml:var>\text{x}_{\text{result}}</ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string">true</owlx:DataValue>
</swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasEffect">
<ruleml:var>\text{PResult}</ruleml:var>
<ruleml:var>\text{expr}_{\text{effect}}</ruleml:var>
</swrlx:individualPropertyAtom>
</ruleml:body>

<ruleml:head>
<swrlx:individualPropertyAtom swrlx:Property="hasInvariant">
<ruleml:var>\text{Pprocess}</ruleml:var>
<ruleml:var>\text{cond}_{\text{inv}}</ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
<ruleml:var>\text{cond}_{\text{inv}}</ruleml:var>
<ruleml:var>\text{x}_{\text{inv}}</ruleml:var>
</swrlx:datatypePropertyAtom>
<swrlx:builtinAtom swrlx:builtin="&\text{swrlb;#compare}">
<ruleml:var>\text{x}_{\text{inv}}</ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string">true</owlx:DataValue>
</swrlx:builtinAtom>
</ruleml:head>
</ruleml:Implies>
If the condition is true the “ComputedInput” has as value the value of the process specified by the property “then”.

```xml
<ruleml:Assertion>
<ruleml:ForAll>
<ruleml:var type="CompositeProcess"> Scomp </ruleml:var>
<ruleml:And>
<swrlx:individualPropertyAtom swrlx:Property="composedOf">
<ruleml:var> Scomp </ruleml:var>
<ruleml:var> Sif-then-elsecontrolconstruct </ruleml:var>
</swrlx:individualPropertyAtom>
<ruleml:ForAll>
<ruleml:var type="Input"> Pinput </ruleml:var>
<ruleml:Implies>
<ruleml:body>
<swrlx:individualPropertyAtom swrlx:Property="ifCondition">
<ruleml:var> Sif-then-elsecontrolconstruct </ruleml:var>
<swrlx:individualPropertyAtom swrlx:Property="expressionBody">
<ruleml:var> xif-then-else </ruleml:var>
<swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
<ruleml:var> xif-then-else </ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
</swrlx:builtinAtom>
<ruleml:individualPropertyAtom swrlx:Property="then">
<ruleml:var> Pperform </ruleml:var>
</ruleml:individualPropertyAtom>
<ruleml:individualPropertyAtom swrlx:Property="process">
<ruleml:var> Pperform </ruleml:var>
</ruleml:individualPropertyAtom>
<ruleml:individualPropertyAtom swrlx:Property="hasInput">
<ruleml:var> Pprocess </ruleml:var>
</ruleml:individualPropertyAtom>
</ruleml:body>
</ruleml:Implies>
</ruleml:ForAll>
</ruleml:Assertion>
```
In the second case the value is determined by the process of the property “else”.

\[
\begin{align*}
\text{\textless ruleml:var type=\textquote{\textquote{CompositeProcess}}\textgreater } & S_{comp} \text{ \textless ruleml:var } \\
\text{\textless ruleml:var type=\textquote{Input}} & P_{input} \text{ \textless ruleml:var } \\
\text{\textless ruleml:body } & \\
\text{\textless swrlx:individualPropertyAtom swrlx:Property=\textquote{ifCondition} } & S_{if-then-elsecontrolconstruct} \text{ \textless ruleml:var } \\
\text{\textless ruleml:var } & \text{ \textless ruleml:var } \\
\text{\textless swrlx:datatypePropertyAtom swrlx:Property=\textquote{expressionBody} } & S_{if-then-elsecontrolconstruct} \text{ \textless ruleml:var } \\
\text{\textless ruleml:var } & \text{ \textless ruleml:var } \\
\text{\textless swrlx:builtinAtom swrlx:builtin=\textquote{\&swrlb;\#compare} } & x_{if-then-else} \text{ \textless ruleml:var } \\
\text{\textless owlx:DataValue owlx:datatype=\textquote{xsd:string} } & false \text{ \textless owlx:DataValue } \\
\text{\textless swrlx:builtinAtom } & \\
\text{\textless swrlx:individualPropertyAtom swrlx:Property=\textquote{else} } & S_{if-then-elsecontrolconstruct} \text{ \textless ruleml:var } \\
\text{\textless ruleml:var } & P_{perform} \text{ \textless ruleml:var } \\
\text{\textless swrlx:individualPropertyAtom } & \\
\text{\textless swrlx:individualPropertyAtom swrlx:Property=\textquote{process} } & S_{if-then-elsecontrolconstruct} \text{ \textless ruleml:var } \\
\text{\textless ruleml:var } & P_{perform} \text{ \textless ruleml:var } \\
\text{\textless swrlx:individualPropertyAtom } & \\
\text{\textless swrlx:individualPropertyAtom swrlx:Property=\textquote{hasInput} } & S_{if-then-elsecontrolconstruct} \text{ \textless ruleml:var } \\
\text{\textless ruleml:var } & P_{process} \text{ \textless ruleml:var } \\
\text{\textless ruleml:var } & P_{input} \text{ \textless ruleml:var } \\
\text{\textless swrlx:individualPropertyAtom } & \\
\end{align*}
\]
In the case, where the condition is true the “ComputedOutput” has as value the value of the process specified by the property “then”.

Vassiliki Alevizou
If condition is false the “ComputedOutput” satisfied by the process of the property “else”.

University of Crete, Computer Science Department
If the condition is true the property “ComputedPrecondition” of the complex service, has as value the value of the process that specified by the property “then”.

Vassiliki Alevizou
Else, the property value satisfied by the property “hasPrecondition” of the process which referred in the property “else”.

```
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="composedOf">
        <ruleml:var> S_comp </ruleml:var>
        <ruleml:var> S_if-then-elsecontrolconstruct </ruleml:var>
      </swrlx:individualPropertyAtom>
      <ruleml:ForAll>
        <ruleml:var type="Condition"> cond_pre </ruleml:var>
        <ruleml:Implies>
          <ruleml:body>
            <swrlx:individualPropertyAtom swrlx:Property="ifCondition">
              <ruleml:var> S_comp </ruleml:var>
              <ruleml:var> cond_if-then-else </ruleml:var>
            </swrlx:individualPropertyAtom>
            <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
              <owlx:DataValue owlx:datatype="xsd:string"> true </owlx:DataValue>
            </swrlx:datatypePropertyAtom>
            <swrlx:individualPropertyAtom swrlx:Property="then">
              <ruleml:var> S_if-then-elsecontrolconstruct </ruleml:var>
            </swrlx:individualPropertyAtom>
            <ruleml:var> P_perform </ruleml:var>
          </ruleml:body>
        </ruleml:ForAll>
      </ruleml:Implies>
    </ruleml:And>
  </ruleml:ForAll>
</ruleml:Assertion>
```
The same is followed for the property “computedEffect”. If the condition is true, the following rule must be satisfied.

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
    <ruleml:And>
      <swrlx:individualPropertyAtom swrlx:Property="composedOf">
        <ruleml:var> S_comp </ruleml:var>
        <ruleml:var> S_if-then-elsecontrolconstruct </ruleml:var>
      </swrlx:individualPropertyAtom>
      <swrlx:individualPropertyAtom swrlx:Property="hasPrecondition">
        <ruleml:var> Pprocess </ruleml:var>
        <ruleml:var> cond_pre </ruleml:var>
      </swrlx:individualPropertyAtom>
    </ruleml:ForAll>
    <ruleml:Implies>
      <ruleml:And>
        <ruleml:ForAll>
          <ruleml:var type="Expression"> expr_effect </ruleml:var>
          <ruleml:var type="Condition"> cond_result </ruleml:var>
        </ruleml:ForAll>
      </ruleml:And>
    </ruleml:Implies>
  </ruleml:ForAll>
</ruleml:Assertion>
```
<ruleml:body>
  <swrlx:individualPropertyAtom swrlx:Property="ifCondition">
    <ruleml:var> $if-then-elsecontrolconstruct </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:individualPropertyAtom swrlx:Property="expressionBody">
    <ruleml:var> condif-then-else </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:individualPropertyAtom swrlx:Property="expressionBody">
    <ruleml:var> xif-then-else </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
    <ruleml:var> xif-then-else </ruleml:var>
    <owlx::DataValue owlx:datatype="xsd:string"> true </owlx::DataValue>
  </swrlx:builtinAtom>
  <swrlx:individualPropertyAtom swrlx:Property="then">
    <ruleml:var> $if-then-elsecontrolconstruct </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:individualPropertyAtom swrlx:Property="process">
    <ruleml:var> Pperform </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:individualPropertyAtom swrlx:Property="hasResult">
    <ruleml:var> Pprocess </ruleml:var>
    <ruleml:var> Presult </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:individualPropertyAtom swrlx:Property="inCondition">
    <ruleml:var> Presult </ruleml:var>
    <ruleml:var> condresult </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
    <ruleml:var> condresult </ruleml:var>
    <owlx::DataValue owlx:datatype="xsd:string"> true </owlx::DataValue>
  </swrlx:builtinAtom>
  <swrlx:individualPropertyAtom swrlx:Property="hasResult">
    <ruleml:var> Presult </ruleml:var>
    <ruleml:var> exprresult </ruleml:var>
  </swrlx:individualPropertyAtom>
  <swrlx:individualPropertyAtom swrlx:Property="computedEffect">
    <ruleml:var> Scomp </ruleml:var>
    <ruleml:var> exprresult </ruleml:var>
  </swrlx:individualPropertyAtom>
</ruleml:body>

<ruleml:head>
  <swrlx:individualPropertyAtom swrlx:Property="computedEffect">
    <ruleml:var> Scomp </ruleml:var>
    <ruleml:var> exprresult </ruleml:var>
  </swrlx:individualPropertyAtom>
</ruleml:head>
</ruleml:Implies>
</ruleml:ForAll>
Else, if condition is false, the value of the property “hasEffect” is determined by the following rule.

```xml
<ruleml:Assertion>
<ruleml:ForAll>
<ruleml:var type="CompositeProcess"> Scomp </ruleml:var>
<ruleml:And>
<swrlx:individualPropertyAtom swrlx:Property="composedOf">
<ruleml:var> Scomp </ruleml:var>
<ruleml:var> Sif-then-elsecontrolconstruct </ruleml:var>
</swrlx:individualPropertyAtom>
<ruleml:ForAll>
<ruleml:var type="Expression"> exprffect </ruleml:var>
<ruleml:var type="Condition"> condresult </ruleml:var>
<ruleml:Implies>
<ruleml:body>
<swrlx:individualPropertyAtom swrlx:Property="ifCondition">
<ruleml:var> Sif-then-elsecontrolconstruct </ruleml:var>
<ruleml:var> condif-then-else </ruleml:var>
<swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
<ruleml:var> condif-then-else </ruleml:var>
<ruleml:var> xif-then-else </ruleml:var>
<swrlx:builtinAtom swrlx:builtin="&swrlb;#compare">
<ruleml:var> xif-thn-else </ruleml:var>
<owlx:DataValue owlx:datatype="xsd:string"> false </owlx:DataValue>
<swrlx:builtinAtom>
<swrlx:individualPropertyAtom swrlx:Property="else">
<ruleml:var> Sif-then-elsecontrolconstruct </ruleml:var>
<ruleml:var> Pperform </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:Property="process">
<ruleml:var> Pperform </ruleml:var>
<ruleml:var> Pprocess </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:Property="hasResult">
<ruleml:var> PProcess </ruleml:var>
<ruleml:var> Presult </ruleml:var>
</swrlx:individualPropertyAtom>
<swrlx:individualPropertyAtom swrlx:Property="inCondition">
<ruleml:var> Presult </ruleml:var>
<ruleml:var> condresult </ruleml:var>
</swrlx:individualPropertyAtom>
</swrlx:datatypePropertyAtom>
</swrlx:builtinAtom>
</swrlx:individualPropertyAtom>
</swrlx:individualPropertyAtom>
</ruleml:body>
</ruleml:Implies>
</ruleml:var>
</ruleml:Assertion>
</ruleml:ForAll>
</ruleml:And>
</ruleml:Assertion>
```
Finally, the same is followed for the property “computedInvariant”. The rules that specify the value of this property in each of two cases are presented below.

```xml
<ruleml:Assertion>
  <ruleml:ForAll>
    <ruleml:var type="CompositeProcess"> S_comp </ruleml:var>
  </ruleml:ForAll>
  <ruleml:And>
    <swrlx:individualPropertyAtom swrlx:Property="composedOf">
      <ruleml:var> S_comp </ruleml:var>
    </swrlx:individualPropertyAtom>
    <ruleml:ForAll>
      <ruleml:var type="Condition"> cond_inv </ruleml:var>
    </ruleml:ForAll>
    <ruleml:Implies>
      <ruleml:body>
        <swrlx:individualPropertyAtom swrlx:Property="ifCondition">
          <ruleml:var> S_if-then-elsecontrolconstruct </ruleml:var>
        </swrlx:individualPropertyAtom>
        <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
          <ruleml:var> cond_if-then-else </ruleml:var>
        </swrlx:datatypePropertyAtom>
      </ruleml:body>
      <swrlx:datatypePropertyAtom swrlx:Property="expressionBody">
        <ruleml:var> xif-then-else </ruleml:var>
      </swrlx:datatypePropertyAtom>
    </ruleml:Implies>
  </ruleml:And>
</ruleml:Assertion>
```
<swrl:builtinAtom swrl:builtin="&swrlb;#compare">  
<ruleml:var>xif-then-else</ruleml:var>  
<owlx:DataValue owlx:datatype="xsd:string">true</owlx:DataValue>  
</swrl:builtinAtom>  
<swrlx:individualPropertyAtom swrlx:Property="then">  
<ruleml:var>Sif-then-else</ruleml:var>  
<ruleml:var>Pperform</ruleml:var>  
</swrlx:individualPropertyAtom>  
<swrlx:individualPropertyAtom swrlx:Property="process">  
<ruleml:var>Pperform</ruleml:var>  
<ruleml:var>Pprocess</ruleml:var>  
</swrlx:individualPropertyAtom>  
<swrlx:individualPropertyAtom swrlx:Property="hasInvariant">  
<ruleml:var>Pprocess</ruleml:var>  
<ruleml:var>condinv</ruleml:var>  
</swrlx:individualPropertyAtom>  
</ruleml:body>  
<ruleml:head>  
<swrlx:individualPropertyAtom swrlx:Property="computedInvariant">  
<ruleml:var>Scomp</ruleml:var>  
<ruleml:var>condinv</ruleml:var>  
</swrlx:individualPropertyAtom>  
</ruleml:head>  
</ruleml:Implies>  
</ruleml:ForAll>  
</ruleml:And>  
</ruleml:ForAll>  
</ruleml:Assertion>
\[ \text{if-then-else} \quad \text{else} \quad \text{perform} \quad \text{process} \quad \text{hasInvariant} \quad \text{computedInvariant} \]

\[ \text{cond}_{\text{inv}} \quad \text{cond}_{\text{inv}} \quad \text{cond}_{\text{inv}} \]

\[ \text{false} \]

\[ \text{Sif-then-else} \quad \text{Pperform} \quad \text{Pprocess} \quad \text{Pprocess} \quad \text{cond}_{\text{inv}} \quad \text{Scomp} \quad \text{cond}_{\text{inv}} \quad \text{cond}_{\text{inv}} \]

\[ \text{false} \]

\[ \text{else} \quad \text{perform} \quad \text{process} \quad \text{hasInvariant} \quad \text{computedInvariant} \]

\[ \text{cond}_{\text{inv}} \quad \text{cond}_{\text{inv}} \quad \text{cond}_{\text{inv}} \]

\[ \text{false} \]