Chapter 6
Semantic Web Services Fundamentals*

Stijn Heymans, Jörg Hoffmann, Annapaola Marconi, Joshua Phillips, and Ingo Weber

Abstract The research area of Semantic Web Services investigates the annotation of services, typically in a SOA, with a precise mathematical meaning in a formal ontology. These annotations allow a higher degree of automation. The last decade has seen a wide proliferation of such approaches, proposing different ontology languages, and paradigms for employing these in practice. The next chapter gives an overview of these approaches. In the present chapter, we provide an understanding of the fundamental techniques, from Artificial Intelligence and Databases, on which they are built. We give a concise, ontology-language independent, overview of the techniques most frequently used to automate service discovery and composition.

* In Alistair Barros, Daniel Oberle (eds.): *Handbook of Service Description: USDL and its Methods*, Part I, Chapter 6, pages 135–158. Springer, New York, 2011.

Stijn Heymans, Joshua Phillips
SemanticBits, 13921 Park Center Road, Suite 420, Herndon, VA 20171, USA, e-mail: stijn.heymans@semanticbits.com, e-mail: joshua.philipps@semanticbits.com

Jörg Hoffmann
INRIA Nancy – Grand Est, Equipe MAIA, 615 rue du Jardin Botanique, 54600 Villers-les-Nancy, France, e-mail: joerg.hoffmann@inria.fr

Annapaola Marconi
Fondazione Bruno Kessler, via Sommarive 18, 38121, Povo, Trento, Italy,
e-mail: marconi@fbk.eu

Ingo Weber
The University of New South Wales, School of Computer Science & Engineering, K17, The University of New South Wales, Sydney, NSW 2052, Australia,
e-mail: ingo.weber@cse.unsw.edu.au

6.1 Introduction

Besides the SOA approach just discussed, a second thread of research on service descriptions and their exploitation is the field of *Semantic Web Services*. The basic idea is to describe services in the context of the Semantic Web, annotating them with a description using a formal ontology to express their precise mathematical meaning. This enables rich support for handling services, support that is not possible based on less information-rich descriptions such as WSDL. In other words, the ontological annotation serves to "explain" the services in more formal detail, and these details allow a higher degree of automation.

The Semantic Web Services research area started in the early 2000s, amongst others with the seminal (and emblematic) paper by McIlraith et al. [53]. As presented in that paper, the main goal of Semantic Web Services approaches is the automation of service discovery and service composition in a SOA. The literature of the last decade has seen a wide proliferation of such approaches. These differ in terms of the ontology languages proposed — there is a wide range of possible formalizations and implementations (language syntaxes) — and in terms of the paradigms proposed for employing these in practice. Chapter 7 is dedicated to providing an overview of these approaches, outlining their commonalities and differences. The present chapter provides an understanding of the fundamental techniques on which they are built. These techniques are drawn from a range of research areas, prominently including Artificial Intelligence and Databases.

We give a concise overview of the techniques most frequently used to automate service discovery and composition. To make this accessible, most of our discussion is informal. Where it is formal, we base it on simple mathematical notations common in the respective areas, thus disregarding the intricacies of practical ontology languages and their implementations.

We cover *Description Logics*, *Logic Programming*, *Planning for Service Chaining*, and *Planning for Service Interactions*. Table 6.1 provides an overview of these, along with their basic distinguishing properties.

Table 6.1: Survey of techniques underlying Semantic Web Services.

Approach	Purpose	Annotation: What is annotated?	Annotation: In which form?	Output of Approach
Description Logics	Discovery	WS input/output	1 logical formula	Set of WS matching query
Logic Programming	Discovery	WS input/output	1 logical formula	Set of WS matching query
Planning for Service Chaining	Composition	Input&prerequisites, output&effects	2 logical formulas	WS chain (composition template)
Planning for Service Interactions	Composition	WS Interface	State transition system	Executable WS composition

Discovery is similar to Web search: given a discovery query, the technology supports the detection of a (potentially ranked) subset of Web services matching the query. Description Logics and Logic Programming offer to improve the precision

and recall of such discovery by formulating the query in a more precise way than with keywords. The discovery query will state, formalized in the respective logic, the kind of input the user can provide to the service, and the kind of output the user expects from the service. This query will be matched against Web services whose input/output is annotated in the same logic, thus allowing to find the subset of services relying on the available input, and delivering the desired output.

Composition is a more complex task, where we not only wish to find a suitable Web service, but where we wish to create, using a subset of already available Web services as atomic building blocks, a more complex Web service providing a more useful functionality. This is a form of programming, thus doing it automatically is quite a challenge. Planning for Service Chaining relaxes this challenge by viewing Web services as one-shot applications, taking into account their input/output typing and high-level properties (such as, available credit), but ignoring technical details such as their interaction patterns and any data transformations needed. Thus the approach provides only a composition template, pre-selecting and arranging a subset of relevant Web services. Planning for Service Interactions tackles the composition challenge in full, delivering an executable software artefact. Accordingly, the approach requires more detailed annotations, involving in particular a specification of how to interact with the Web service. This comes in the form of a transition system, i.e., a kind of abstract program similar to BPEL abstract processes [58].

In what follows, we focus in turn on Description Logics (Section 6.2), Logic Programming (Section 6.3), Planning for Service Chaining (Section 6.4), and Planning for Service Interactions (Section 6.5). For each, we provide a detailed summary at a non-technical level; some technical details are given in separate sub-sections that the reader not interested in such details may skip. We illustrate the approaches using examples, taken from applications where suitable.[3]

6.2 Description Logics

Description logics (DLs) is the most prominent formalization underlying the Semantic Web, and Semantic Web Services. In what follows, we first provide an approach synopsis giving the main facts in an informal way, then we include a detailed treatment of the basic DL formalities. The reader not interested in technical details may skip the latter sub-section.

6.2.1 Approach Synopsis

Description Logics (DLs) are a family of logical formalisms widely used for knowledge representation, e.g., for the representation of terminologies in application do-

[3] Since the approaches are quite different in their underlying intention and scope, there is no one unifying example suitable to illustrate them all.

mains such as healthcare — witnessed by terminologies such as OpenGALEN[4] and SNOMED CT®.[5] Its basic language features include the notions of *concepts* and *roles* which are used to define the relevant concepts and relations in some (application) domain. Different DLs can then be identified, among others, by the set of constructors that are allowed to form complex concepts or roles.

The combination of a formal well-understood semantics and the availability of practical reasoners,[6] has led to the adoption of DLs as the formal underpinning of ontology languages such as OWL [59] on the Semantic Web or in the use of Semantic Web Services [53]. In the context of Semantic Web Services they are used with different purposes, e.g., to express the background domain ontologies, as the language for pre- and post-condition, input and output descriptions, ...

We give a small example showing the benefits of using (simple) Description Logics in the context of Web services. Further note that we make the simplifying assumption that Web services have one operation and that when given an input, they just give an output (no choreography).

Consider two services *S1-cure* and *S2-cause*. *S1-cure* takes as input a particular *Allergy* (one can see this as a SNOMED CT® concept) and returns as output a *Substance* (again a SNOMED CT® concept) that can alleviate the symptoms of the allergy. We write this as follows (using notation from [42]):

S1-cure:
INPUT *x Allergy*
OUTPUT *y Substance*

In words, given an allergy *x*, return the substance *y* that could resolve its symptoms. Note that *Allergy* and *Substance* are simple DL concept names. The service *S2-cause* takes exactly the same input and output but returns a substance that could be a cause of the allergy symptoms.

S2-cause:
INPUT *x Allergy*
OUTPUT *y Substance*

Two immediate issues arise:

1. Assume a user has a request Q for a service that inputs a certain allergy and would like to know a possible cause. Both Web services (described by WSDL if you want, where *Allergy* and *Substance* would be specific message types), are described in identical ways but do 2 entirely different things. Only *S2-cause* would be a suitable Web service satisfying the user's request, but both *S1-cure* and *S2-cause* would be returned as suitable Web services for the user based on the input and output types. Note that according to [42] this occurs commonly in the biomedical domain where services often have as input and output just strings.

[4] `http://www.opengalen.org/`

[5] `http://www.ihtsdo.org/snomed-ct/`

[6] Several reasoners for expressive DLs have emerged since the 80s, e.g., Racer [34], FaCT [41], Pellet [70], and HermiT [69].

2. Assume a user has a request Q that involves finding a service that takes a penicillin allergy as input and gives the resolving substances. Even though every penicillin allergy is an allergy and thus *S1-cure* would be able to resolve the issue, it would not match as the input type of *S1-cure* does not correspond to the input of the request.

Using a background domain ontology that indicates that *PCNAllergy* is a subclass of *Allergy* (as does SNOMED CT®), in addition to the above descriptions of *S1-cure* and *S2-cause*, would rightfully propose *S1-cure* as a solution to the request Q where Q is as follows:

Q:
INPUT *x PCNAllergy*
OUTPUT *y Substance*

Indeed, every *PCNAllergy* would be an *Allergy* according to the ontology, such that *S1-cure* which takes allergies as input could propose substances. Formally, a statement "every *PCNAllergy* is a *Allergy*" is called a *DL (subclass) axiom*, similar to a simple *is-a* specification in conceptual modeling.

Note that also *S2-cause* is applicable: the knowledge that the user needs a resolution and not a causing substance is not made explicit yet (neither is it explicit that *S1-cure* provides a curing substance nor that *S2-cause* provides a causing substance). Making this explicit can be done by pre- and post-conditions.

6.2.2 AI Formalism

We introduce a canonical version of a Description Logic; the reader can easily skip this technical section if need be.

The semantics of DLs is given by first-order style interpretations $\mathscr{I} = (\Delta^{\mathscr{I}}, \cdot^{\mathscr{I}})$ where $\Delta^{\mathscr{I}}$ is a non-empty domain and $\cdot^{\mathscr{I}}$ is an interpretation function. The basic building blocks are concept names, role names (abstract or concrete, possibly inverted in case of the former), data types, and nominals. For more details, we refer the reader to [8].

Based on those building blocks, we define *concept expressions* as in Table 6.2, where A is a concept name, R, S are abstract roles, T is a concrete role name, $d \in \mathbf{D}$ is a data type, and C, D are concept expressions.

A DL *knowledge base* is a set of *axioms*, where an axiom is of one of the following three types, respectively indicating subset relations between concept expressions, subset relations between roles, and transitivity of roles.

- *terminological axioms* $C \sqsubseteq D$ with C and D concept expressions,
- *role axioms* $R \sqsubseteq S$ where R, S may be inverse roles with the underlying roles both abstract or both concrete, and
- *transitivity axioms* $\mathsf{Trans}(R)$ for an (inverse) abstract role.

Table 6.2: Syntax and Semantics of DL Constructs

Construct name	Syntax	Semantics
concept conj.	$C \sqcap D$	$(C \sqcap D)^{\mathscr{I}} = C^{\mathscr{I}} \cap D^{\mathscr{I}}$
concept disj.	$C \sqcup D$	$(C \sqcup D)^{\mathscr{I}} = C^{\mathscr{I}} \cup D^{\mathscr{I}}$
negation	$\neg C$	$(\neg C)^{\mathscr{I}} = \Delta^{\mathscr{I}} \setminus C^{\mathscr{I}}$
exists restriction	$\exists R.C$	$(\exists R.C)^{\mathscr{I}} = \{x \mid \exists y : (x,y) \in R^{\mathscr{I}} \text{ and } y \in C^{\mathscr{I}}\}$
value restriction	$\forall R.C$	$(\forall R.C)^{\mathscr{I}} = \{x \mid \forall y : (x,y) \in R^{\mathscr{I}} \Rightarrow y \in C^{\mathscr{I}}\}$
atleast restriction	$\geq nS.C$	$(\geq nS.C)^{\mathscr{I}} = \{x \mid \#\{y \mid (x,y) \in S^{\mathscr{I}} \text{ and } y \in C^{\mathscr{I}}\} \geq n\}$
atmost restriction	$\leq nS.C$	$(\leq nS.C)^{\mathscr{I}} = \{x \mid \#\{y \mid (x,y) \in S^{\mathscr{I}} \text{ and } y \in C^{\mathscr{I}}\} \leq n\}$
data type exists	$\exists T.d$	$(\exists T.d)^{\mathscr{I}} = \{x \mid \exists y : (x,y) \in T^{\mathscr{I}} \text{ and } y \in d^{\mathbf{D}}\}$
data type value	$\forall T.d$	$(\forall T.d)^{\mathscr{I}} = \{x \mid \forall y : (x,y) \in T^{\mathscr{I}} \Rightarrow y \in d^{\mathbf{D}}\}$

Traditionally, a knowledge base contains also assertional statements[7] such as $C(a)$ (or $R(a,b)$) which intuitively means that the individual a is an instance of C (a is related to b by means of the role R).

Terminological and role axioms express a subset relation: an interpretation $\mathscr{I}$ *satisfies* an axiom $C_1 \sqsubseteq C_2$ ($R_1 \sqsubseteq R_2$) if $C_1^{\mathscr{I}} \subseteq C_2^{\mathscr{I}}$ ($R_1^{\mathscr{I}} \subseteq R_2^{\mathscr{I}}$). An interpretation satisfies a transitivity axiom $\mathsf{Trans}(R)$ if $R^{\mathscr{I}}$ is a transitive relation. An interpretation is a *model* of a knowledge base Σ if it satisfies every axiom in Σ. A concept C is *satisfiable* w.r.t. Σ if there is a model $\mathscr{I}$ of Σ such that $C^{\mathscr{I}} \neq \emptyset$.

As indicated above, DLs are useful for expressing knowledge in the healthcare domain. For example, the *Systematized Nomenclature of Medicine–Clinical Terms* (SNOMED CT®) [1] is a reference terminology for clinical data that can be seen as a particular DL knowledge base.

Example 6.1. Consider the example knowledge base Σ in Table 6.3, loosely inspired by SNOMED CT®.

Table 6.3: SNOMED CT® Fragment Amoxicillin

(1)	$\mathit{AmoxicillinTablet} \sqsubseteq \exists \texttt{hasActiveIngredient}.\mathit{Amoxicillin}$
(2)	$\mathit{Amoxicillin} \sqsubseteq \mathit{Penicillin}$
(3)	$\mathit{SafeForPCNAllergies} \sqsubseteq \forall \texttt{hasActiveIngredient}.\neg \mathit{Penicillin}$

The example indicates in axiom (1) that amoxicillin tablets have an active ingredient that is an amoxicillin, which in turn is a penicillin by (2). According to (3), we collect in the concept *SafeForPCNAllergies* all elements that have only active ingredients that are not penicillins. A possible interpretation is $\mathscr{I}$ with $\Delta^{\mathscr{I}} = x, y$ and

[7] The assertional statements in a knowledge base are also referred to as the *ABox*, while the non-assertional statements, the *terminological* statements, are referred to as the *TBox*.

with $AmoxicillinTablet^{\mathscr{I}} = x$, $\texttt{hasActiveIngredient}^{\mathscr{I}} = (x,y)$, $Amoxicillin^{\mathscr{I}} = Penicillin^{\mathscr{I}} = y$, $SafeForPCNAllergies = \emptyset$. This interpretation is clearly a model of Σ. If we are interested in knowing whether amoxicillin tablets are safe to take when you have a penicillin allergy, one could check whether $\Sigma \models AmoxicillinTablet \sqsubseteq SafeForPCNAllergies$, i.e., does *SafeForPCNAllergies* subsumes *AmoxicillinTablet* w.r.t. Σ. As we have a model $\mathscr{I}$ where $AmoxicillinTablet^{\mathscr{I}} \not\subseteq SafeForPCNAllergies^{\mathscr{I}}$, we can answer this negatively.

We could use such reasoning over a domain ontology (SNOMED CT®) in the context of discovery of Web services. For example, a user goal might be to find all services that output medication that is safe for patients with penicillin allergies. A service that has its output modeled using the concept *SafeForPCNAllergies* would satisfy this exactly. One can for example deduce using standard DL reasoning that services that output Amoxicillin tablets are not matching this goal (as *AmoxicillinTablet* is not subsumed by *SafeForPCNAllergies*) above.

6.3 Logic Programming

Logic Programming (LP) is the main alternative to DL, for handling Semantic Web Services. As before, we first provide an approach synopsis, then include a detailed treatment of the basic LP formalities (which the reader not interested in technical details may skip).

6.3.1 Approach Synopsis

Whereas expressing knowledge in Description Logics revolves around atomic concepts and roles to construct more expressive concept expressions, Logic Programming traditionally has as its basic building blocks first-order atoms. In particular, writing a DL-concept *4Door* as an atom, results in $4Door(X)$ where X is a variable and *4Door* is in that context a predicate for example identifying an item X as a 4-door car. Similarly, we have that DL roles such as *car_available* correspond to atoms $car\_available(X,Y)$ indicating that rental company X has a car Y available.

Whereas traditional DLs have only concepts and roles as basic building blocks, LP allows usually as well for n-ary atoms such as $travels(From,To,Name)$ indicating that *Name* travels from *From* to *To*.

Combining these building blocks in DLs is done by defining syntactical structures such as exist restrictions $\exists car\_available.4Door$ (the members of which all have a 4-door car available); in LP on the other hand we combine atoms by simple conjoining or disjoining them. For example, $car\_available(avis,X), 4Door(X)$ indicates that *avis* has some (X) car available that is a 4-door car.

Actually expressing knowledge is then done very similarly in LP as in DLs by means of an IF-THEN structure (recall the example in the previous section that an

PCNAllergy is-a particular *Allergy*):

$$rentAt(X) \leftarrow car\_available(X,Y), 4Door(X)$$

which indicates that if X has a 4-door car Y available, then one wants to rent at X. Note that such rules would be typical rules in a Virtual Travel Agency discovery scenario to express the preferences of a prospective renter.

In the presence of negation, different semantics have been proposed historically for such sets of rules (viz., *logic programs*), two of the prominent ones being *well-founded* semantics [28] and the *answer set semantics* [29].

Logic Programming, in particular the answer set methodology has been successfully applied in problem areas such as planning [47], configuration and verification [72], diagnosis [22], and database repairs [6]. Moreover, several *answer set solvers*, i.e., systems that return the answer sets of the program, have reached a mature stage of development. E.g., SMODELS [57] and DLV [46].

Moreover, as the envisioned basis of future information systems, the Semantic Web is a fertile ground for deploying AI techniques, and in turn raises new research problems in AI. As a prominent example, the combination of rules with Description Logics (DLs), which is central to the Semantic Web architecture, has received high attention over the past years, with approaches such as *Description Logic Programs* [33], *DL-safe rules* [56], *r-hybrid KBs* [67], $\mathcal{DL}$*+log* [68], *MKNF KBs* [55], *Description Logic Rules and ELP* [43], and *dl-programs* [24].

In the area of Semantic Web Services, Logic Programming plays an important role in for example the Web Service Modeling Language (WSML), [21]. As the backbone of WSML it plays a similar role as Description Logics, the backbone of OWL-DL. Indeed, it is used in the expression of background ontologies, the expression of goals, pre-conditions, post-condition, capabilities of services etc.

We summarize some differences between the paradigms of Description Logics and Logic Programming in Table 6.4. Note that there a lot of variations of the both, so Table 6.4 should be seen more as the general case than as covering all cases.

- Logic Programming in general has a minimal model semantics. In other words, one is interested in the minimal model (w.r.t. the subset relation) of a set of rules.
- A consequence of the minimal model semantics for LP, is that reasoning in LP is *nonmonotonic* vs *monotonic* in DLs. In Logic Programming, entailments of a logic program might not hold any longer after rules or facts where added to that logic program.
- Logic programming has a *closed domain assumption*: only the constants appearing in a logic program are relevant for the construction of models. This in contrast with Description Logics, which are generally speaking a fragment of first-order logic and have an *open domain assumption* where any non-empty domain can potentially serve as the universe/domain of the knowledge base.

Table 6.4: Differences Description Logics and Logic Programming.

DL	LP
model semantics	minimal model semantics
monotonic	nonmonotonic
open domain	closed domain

6.3.2 AI Formalism

In order to make the exposition as simple as possible, we define the language of answer set programming as a canonical example of a logic programming paradigm; the reader can easily skip this technical section if need be.

Note that in the case of logic programs without negation, the answer set semantics coincides with the canonical minimal model semantics, and in the case of stratified logic programs, the well-founded semantics coincides with the answer set semantics.

Terms, atoms, literals are defined as usual (see [9] for details); an *extended literal* is a literal l or a *naf-literal* not l, i.e., a literal preceded with the *negation as failure* symbol *not*.

A *(logic) program (LP)* is a countable set of *rules* $\alpha \leftarrow \beta$, where α and β are finite sets of extended literals, respectively called the *head* and *body* of the rule. The body of a rule is considered to be a conjunction of extended literals (denoted as a comma-separated list) and the head as a disjunction of extended literals (denoted as a $\vee$-separated list). The *positive part* of a set of extended literals γ is $\{\gamma^+ \equiv l \mid l \in \gamma, l \text{ literal}\}$ and the *negative part* is $\{\gamma^- \equiv l \mid not\ l \in \gamma\}$.

A *ground* atom, (extended) literal, rule, or program does not contain variables.

All following definitions in this section assume ground programs and ground (extended) literals. answer set of $gr(P)$. The *Herbrand Base* $\mathscr{B}_P$ of a program P is the set of all ground atoms that can be formed using the language of P. An *interpretation* I of P is any consistent subset of $\mathscr{L}_P$, where $\mathscr{L}_P$ is the set of all ground literals that can be formed using the language of P, i.e., $\mathscr{L}_P = \mathscr{B}_P \cup \neg\mathscr{B}_P$. For a literal l, we write $I \models l$, if $l \in I$, which extends for extended literals *not* l to $I \models not\ l$ if $I \not\models l$. In general, for a set of extended literals X, $I \models X$ if $I \models x$ for every extended literal $x \in X$. A rule $r : \alpha \leftarrow \beta$ is *satisfied* w.r.t. I, denoted $I \models r$, if $\exists l \in \alpha I \models l$, for some extended literal l, whenever $I \models \beta$, i.e., r is *applied* ($\exists l \in \alpha I \models l$ and $I \models \beta$) whenever it is *applicable* ($I \models \beta$). The set of satisfied rules in P w.r.t. I is the *reduct* P_I.

For a *simple* program P (i.e., a program without *not*), an interpretation I is a *model* of P if I satisfies every rule in P, i.e., $P_I = P$; it is an *answer set* of P if it is a minimal model of P, i.e., there is no model J of P such that $J \subset I$. We define answer sets for programs with *not* in terms of a reduction to simple programs. The *GL-reduct*[8] w.r.t. an interpretation I is the simple P^I, where P^I contains $\alpha^+ \leftarrow \beta^+$

[8] Named after its inventors M. Gelfond and V. Lifschitz.

for $\alpha \leftarrow \beta$ in P, $I \models \alpha^-$, and $I \models not\ \beta^-$. Thus, given an interpretation I of literals — the items that one supposes true — the GL-reduct contains those rules for which the negative part is consistent with the beliefs in I. If there is a naf-literal in the body that is not true in I, then the rule is not in the GL-reduct since its whole body is then false and cannot be used to deduce literals. If all naf-literals in the body are true, the rule stays in the GL-reduct (depending on the naf-literals in the head), but with the naf-literals removed (they are known to be true). A similar reasoning holds for the head of a rule: if there is a naf-literal in the head that is true w.r.t. I, we have that the rule is automatically true and can be removed; if all naf-literals in the head are false, then we remove them and leave the rule in the GL-reduct.

I is an *answer set* of P if I is an answer set of P^I. Thus, given an interpretation I, one calculates the GL-reduct, and checks that the minimal model of the GL-reduct is I; an answer set is thus *self-motivating* or *stable*.

For more details, we refer to [9, 20].

6.4 Planning for Service Chaining

The AI formalism we review now allows to describe services in terms of "preconditions" and "effects," serving to automatically compose service chains respecting (amongst possibly other things) the input and output behavior of the services.

The following sub-sections provide: a brief formalism summary; a more detailed description of the formalism (the reader not interested in technical details may skip this sub-section); a summary of the application to service composition; an example application; and a brief discussion of variants and their merits.

6.4.1 Formalism in a Nutshell

Planning is one of the long-standing sub-areas of AI, originating in the 1960s. In a nutshell, the approach allows the user to describe, in a high-level language, a problem involving an *initial state*, a *goal*, and a set of *actions*. The AI tool — the *planning system* — then automatically finds a schedule of actions — the *plan* — transforming the initial state into a goal state.

Actions are described in terms of two logical formulas, the *precondition* and the *effect*. The former states the condition that must hold for the action to be applicable, in a given state. The latter states the condition that holds after the action has been applied, i.e., it specifies how the action changes the state.

6.4.2 Formalism Details

Over the past four decades, the planning literature has come up with a plethora of frameworks for planning. For a comprehensive introduction into the field, we recommend the recent book by Ghallab et al. [31]. We describe in what follows one of the simplest planning formalisms, called *planning with finite-domain variables* [35]. A brief discussion of the wider literature follows below.

A finite-domain variable *planning task* is a 4-tuple (X, s_0, s_*, O). X is a finite set of *variables*, where each $x \in X$ is associated with a finite domain D_x. A *partial state* over X is a function s on a subset X_s of X, so that $s(x) \in D_x$ for all $x \in X_s$; s is a *state* if $X_s = X$. The *initial state* s_0 is a state. The *goal* s_* is a partial state. O is a finite set of *operators*. Each $o \in O$ is a pair $o = (\text{pre}_o, \text{eff}_o)$ of partial states, called its *precondition* and *effect*. Partial states are identified with sets of variable-value pairs, referred to as *facts*. The *state space* of the task is the directed graph whose vertices are all states over X, with an arc (s, s') iff there exists $o \in O$ such that $\text{pre}_o \subseteq s$, $\text{eff}_o \subseteq s'$, and $s(x) = s'(x)$ for all $x \in X \setminus X_{\text{eff}_o}$. A *plan* is a path in the state space, leading from s_0 to a state s with $s_* \subseteq s$.

Illustrating this with a simple example, say that we have a variable expressing the status of a flight booking, *Pending* vs. *Confirmed*. The booking is currently pending, we wish it to be confirmed, and the only operator we have is a service confirming the booking. Expressing this in the above formalism, we get: $X = \{\textit{flightStatus}\}$ with $D_{\textit{flightStatus}} = \{\textit{Pending}, \textit{Confirmed}\}$; $s_0 = \{(\textit{flightStatus}, \textit{Pending})\}$; $s_* = \{(\textit{flightStatus}, \textit{Confirmed})\}$; O contains a single operator taking the form $(\{(\textit{flightStatus}, \textit{Pending})\}, \{(\textit{flightStatus}, \textit{Confirmed})\})$ where $\{(\textit{flightStatus}, \textit{Pending})\}$ is the precondition and $\{(\textit{flightStatus}, \textit{Confirmed})\}$ is the effect. The state space contains the two vertices $\{(\textit{flightStatus}, \textit{Pending})\}$ and $\{(\textit{flightStatus}, \textit{Confirmed})\}$, the only arc going from the former to the latter. The plan traverses that arc, thus confirming the flight.

Let us match this formalism against the summary given above in Section 6.4.1. "States" are formalized in terms of (state) variables, where an assignment to all variables defines the state. Note that the number of states is exponential in the number of variables. Thus this language allows to describe compactly a large number of possibilities, and deciding whether or not there exists a plan is computationally hard (**PSPACE**-complete). The precondition/effect "formulas" are restricted here to the simplest possible notion of listing a subset of required/effected variable-value pairs. "Plans" are simple sequences of actions mapping the initial state into a state that complies with the list of variable-value pairs given in the goal.

Traditionally, planning formalisms have been rooted in propositional logics, restricting the state variables to be Boolean, having exactly two possible values: *True* and *False*. The simplest and most wide-spread formalism of this kind is called "STRIPS" [26], which is exactly like the finite-domain variable planning tasks above except that all variables are Boolean. Other formalisms, such as *ADL* [60], allow more complex specifications of pre, eff, and s_*, involving conditional effects and arbitrary 1st-order logic formulas (quantifiers ranging over a finite universe).

In the context of the International Planning Competition[9] that has been held biennially since 1998, a common input syntax for planning systems has been defined. This input language is called *PDDL* — Planning Domain Definition Language — and has a range of variants encompassing STRIPS and ADL [52], numeric and temporal planning [27], and a number of other extensions [38, 30].

Finally, it is relevant in our context to mention the field of *planning under uncertainty* (e.g., [17, 73]). The above formalisms all assume perfect knowledge about the initial state, and deterministic behavior of actions. These assumptions often do not hold in real-world applications, including many applications of service composition. Planning under uncertainty relaxes these assumptions in a variety of ways, of course at the cost of increased (theoretical and practical) computational costs.

6.4.3 Application to Services

The application of planning to service composition has first appeared as an idea in the late 1990s (e.g., [13]) and has been intensively researched since the early 2000s (e.g., [54, 64, 19, 44, 63, 48, 37, 40]). The different approaches differ widely in intention and scope, as well as underlying formalisms (see some details in Section 6.4.5 below). The lowest common denominator is that preconditions/effects allow to specify service behavior at an abstract level where they are understood as atomic one-shot operations. In the OWL-S service description framework (e.g., [2, 18]), this abstraction level is called the "service profile;" in the WSMO framework (e.g., [25]), it is referred to as the "service capability;" see also Chapter 7.

The service profile/capability encompasses of course its input and output behavior, i.e., the typing of the respective service parameters. As a simple example, the service input may be a customer-data object, and the output may be a reservation-object. One can further express additional prerequisites or consequences that the service may have, in a logics-based representation of the relevant surroundings. A precondition "$credit \geq 500$" may require the sufficient availability of money to cover at least the cost of 500 money units, and an effect "$credit := credit - 500$" may reduce the available money by that amount.

Planning is computationally hard in general, but AI research has come up with a range of rather effective approaches (e.g., [39, 65]). Provided that the plans to be created are not too large (up to 20 or so actions), practical runtime/memory performance when using these techniques typically is not a big issue, plans being found within a matter of seconds (e.g., [37, 40]).

A plan in this setting — some scheduling of the atomic services — is not necessarily an executable software artifact. This is because we do not take into account many technical aspects of the services involved. Most importantly, we disregard the order of interactions required for communicating with them (making a reservation typically involves several steps, rather than being a one-shot interaction), and we

[9] See `http://ipc.icaps-conference.org/`

disregard the actual underlying data structures or WSDL schemata (what exactly is a "customer-data object"?).

Since plans are not guaranteed to be executable, the planning facility provided typically accomplishes only a part of the service composition design. A typical perspective is that a human user is responsible for the overall design, and uses the planning facility for easing the task of selecting and arranging a useful subset of services from a large services database or the internet (e.g., [3, 40]). Another possibility is to use this abstract planning as a pre-process to more accurate — but more computationally expensive — service composition techniques (as will be described in Section 6.5). This reduces the size of the input to choose from, and thereby the computational resources required to come up with a composition [10].

In order to run the planning facility, services need be described as actions in the first place, and the user needs to enter the initial state and goal of the desired composed service. These are non-trivial tasks which need to be addressed in a clever way in order to keep the modeling overhead at bay. Several frameworks have appeared that allow users to conveniently design the models via user-friendly interfaces (e.g., [66, 32]). An alternative approach [40] suggests to instead exploit pre-existing models of software behavior, thus getting the planning model "for free," respectively sharing the modeling effort with other software design activities. We will now look at this approach in a little more detail, as an application example.

6.4.4 Example

SAP widely employs model-driven software engineering. The more modern developments are designed as service-oriented architectures (SOAs). Some of the models created in their development aim at describing the behavior of service operations. To achieve this, the *Status and Action Management (SAM)* models exist for each of over 400 business objects (BOs). BOs are software objects which have an actual correspondence in common business scenarios, such as, e.g., a customer quote, a sales order, an invoice, etc.

Each BO can have (vast) data structures and offers functionality on this data as service operations in the SOA. SAM models capture the relation between the status of a BO and the actions (operations) it offers: when can these actions be enacted, and how do they affect the BO? Concretely, SAM models consist of a set of finite-domain status variables and a set of actions that describe which status values for which variables are a precondition to an action, and how the status values may change as a result of a service execution. The original purpose of SAM is code generation: code skeletons check if preconditions are fulfilled at runtime, and update the status variables accordingly.

SAM is based on common business terms. For instance, the status variable "approval" may include values such as "approved" and "rejected." Hence, its expressions are understandable to business users. This is key to practical planning-based service composition, where business users describe in terms of SAM what the com-

Table 6.5: A SAM-like example, modeling the behavior of "customer quotes" CQ.

Action name	precondition	effect
Create CQ		(CQ.approval:necessary OR CQ.approval:notNecessary) AND CQ.acceptance:notAccepted AND CQ.archiving:notArchived AND CQ.submission:notSubmitted
CQ Approval	CQ.approval:necessary	CQ.approval:approved OR CQ.approval:rejected
Submit CQ	CQ.archiving:notArchived AND (CQ.approval:notNecessary OR CQ.approval:granted)	CQ.submission:submitted
Mark CQ as Accepted	CQ.archiving:notArchived AND CQ.submission:submitted	CQ.acceptance:accepted
Archive CQ	CQ.archiving:notArchived	CQ.archiving:archived

position should accomplish (what's its initial state and goal). The planning functionality is implemented as a prototypical research extension to the SAP NetWeaver BPM Process Composer.

The only important difference between SAM and the basic formalism introduced in Section 6.4.2 is that effects can be non-deterministic, i.e., the action has one out of a set of possible outcomes. For the business context, it is quite obvious that this is necessary. Any sensible order, booking, check, or approval action will have at least two possible outcomes — positive vs. negative.

For illustration, Table 6.5 gives a SAM-like model for a particular BO called "customer quote (CQ)." For confidentiality reasons, the shown object and model are artificial, i.e., they are *not* contained in SAM as used at SAP. In the figure, by "CQ.*X*:Y" we denote the atomic proposition stating that variable X of the customer query has value Y. We are using a standard planner [39] — modified to appropriately handle SAM's non-deterministic effects — on this input.

For presentation to the user, a simple post-process transforms plans into BPMN diagrams. Figure 6.1 shows a plan in that representation, for the above example. Note in particular the side effect of the create operation (top): approval can be either necessary or not. Depending on the actual outcome, an additional approval step is executed. In turn, this approval step may have a positive or negative outcome. Only in the positive outcome, the process continues by submitting the quote. No procedure is specified for the negative outcome — exception handling is needed in this case, and SAM does not contain any information about what that handling should be, so the planner cannot compose it automatically. Clearly, the plan is also incomplete in that not every customer quote should be approved, submitted, etc. straight after being created. Thus the plan serves merely as a template and needs to be completed by hand.[10]

[10] Note also that the standard process is often implemented in the standard system. The value of service composition here lies in convenient creation of variants of the standard.

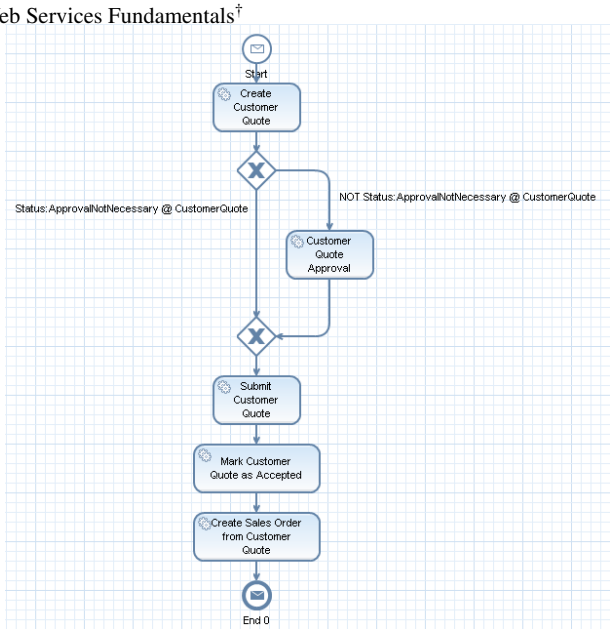


Fig. 6.1: Screenshot of the SAP NetWeaver BPM Process Composer with an automatically composed process of five non-deterministic services).

6.4.5 Discussion

As indicated, the different approaches to planning-based service composition differ widely in intention, scope, and formalization. Let us mention a few of the better-known works. Some authors compile service composition into more or less standard planning formalisms, e.g., [64]. Two frameworks allow the user to provide a plan skeleton whose control structures will be filled in by the planner [54, 71].

A large strand of work addresses the handling of "input/output typing" in a rich Semantic Web framework where the surroundings of the services are described in a formal logics, most often in some variant of Description Logics (DL) as discussed in Section 6.2. A main issue here is that DL contains non-trivial axiomatizations of the domain. These axioms make it difficult to define and compute the "outcome state" resulting from applying an action.[11] One body of work circumvents this problem by applying the axiom inferences only to the service outputs, i.e., to individuals that didn't exist prior to action application ([19, 48, 37]). A few works address the

[11] For example, say the ontology contains the axiom $A \sqsubseteq \neg(B \sqcap C)$ stating (intuitively) that an element of type A cannot be in the intersection of B and C. Say that we are in a state where we have an individual o so that $o \in A$ and $o \in B$. Say we apply an action whose effect is $o \in C$. What is the outcome state? Simply adding $o \in C$ results in a conflict with the axiom. Thus we need to "repair" this outcome state, by removing one of the previous facts, $o \in A$ or $o \in B$. Which one should we remove? This is difficult to answer in a principled manner. Most known answers have severe consequences on the computational complexity of computing outcome states [36].

problem in full, living with the computational costs incurred (e.g., [23]). Yet others address more feasible sub-classes of the problem (e.g., [14]).

A fundamentally different approach is the one described in Section 6.5, whose output is actual executable code [63]. The more abstract approach described in the section at hand can be used as a filtering method in front of process-level composition, reducing the choice of services to compose from, and thus the empirical performance of the more fine-grained composition [10].

In summary, planning for service chaining allows to automatically compose services at an abstract level taking into account their input and output typing, as well as any other prerequisites/effects they have relative to a high-level formalization of their surroundings. The method can be computationally quite feasible, depending on the specifics of the surroundings and on the complexity of any (DL-) axiomatizations that should be taken into account. The outcome of planning is a composition-template which most often is imperfect, but delivers useful input to either humans or other more detailed composition techniques. A key problem in practice is the modeling overhead. Recent work [40] suggests a connection to model-driven software engineering that could be exploited to reduce this overhead significantly.

6.5 Planning for Service Interactions

The AI planning approach presented in this section deals with the problem of automatically composing an executable Web service that, interacting with a set of component services, satisfies a given composition requirement.

Similarly to Section 6.4, the following sub-sections provide: a brief formalism summary; a more detailed description of the formalism (the reader not interested in technical details may skip this sub-section); a summary of the application to service composition; an example application; and a brief discussion of variants and their merits.

6.5.1 Formalism in a Nutshell

The approach is based on planning as model checking [15, 16]. It adopts "state transition systems" for the representation of the individual entities to be composed into a plan. This allows to represent stateful processes implementing a complex interaction protocol, exchanging asynchronous messages, and exhibiting a partially observable and non-deterministic behavior. The input to the planning problem is a set of such processes, as well as a "composition requirement" (a combined functionality we wish to achieve). The solution to the planning problem then is a "controller," another state transition system, such that executing the controller results in an orchestration of the processes achieving the composition requirement.

6.5.2 Formalism Details

Each service is encoded as a *state transition system* (STS from now on) $\Sigma = \langle \mathscr{S}, \mathscr{S}^0, \mathscr{I}, \mathscr{O}, \mathscr{R}, \mathscr{L} \rangle$ that can be in one of its possible states $\mathscr{S}$, a subset of which are *initial* $\mathscr{S}^0$, and can evolve to new states as a result of performing some actions. Actions are distinguished in *input actions* $\mathscr{I}$, which represent the reception of messages, *output actions* $\mathscr{O}$, which represent messages sent to external services, and a special action τ, called *internal action*. The action τ is used to represent internal evolutions that are not visible to external services, i.e., the fact that the state of the system can evolve without producing any output, and independently from the reception of inputs. The *transition relation* $\mathscr{R}$ describes how the state can evolve on the basis of inputs, outputs, or of the internal action τ. Finally, a *labeling function* associates to each state the set of properties in $\mathscr{P}rop$ that hold in the state. These properties will be used to define the composition requirements. For a complete description of the translation from Web services (described in terms of their BPEL and WSDL specification) to STS please refer to [49].

The *behavior* of an STS is represented by its set of possible *runs*, i.e., of sequences $s_0, a_0, s_1, a_1, \ldots$ such that $s_0 \in \mathscr{S}^0$ and $(s_i, a_i, s_{i+1}) \in \mathscr{R}$. In general, such runs may be finite or infinite. A run σ is said to be *completed* if it is finite, and if its last state is final. A state $s \in \mathscr{S}$ will be said *reachable* if there exists a run $\sigma = s_0, a_0, \ldots, a_{n-1}, s_n, \ldots$ such that $s_n = s$. We will denote with $Reachable(\Sigma) \subseteq \mathscr{S}$ the set of reachable states of Σ.

The composition problem has two inputs: the formal composition requirement ρ and the set of component services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$. The component services $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ evolve independently, and in fact represent, under a planning perspective, the domain to be controlled. Such domain $\Sigma_{\|}$ is obtained as the first step of the composition, by combining $\Sigma_{W_1}, \ldots, \Sigma_{W_n}$ by means of a *parallel product* operation. The system representing (the parallel evolutions of) the component services $W_1, \ldots, W_n$ is formally defined as $\Sigma_{\|} = \Sigma_{W_1} \parallel \ldots \parallel \Sigma_{W_n}$.

In a composition problem, the composite service is defined as a "controller" Σ_c (also described as a STS), which interacts with the domain Σ, orchestrating the component services. The STS $\Sigma_c \triangleright \Sigma$, describing the behaviors of system $\Sigma = \langle \mathscr{S}, \mathscr{S}^0, \mathscr{I}, \mathscr{O}, \mathscr{R}, \mathscr{L} \rangle$ when controlled by $\Sigma_c = \langle \mathscr{S}_c, \mathscr{S}_c^0, \mathscr{O}, \mathscr{I}, \mathscr{R}_c, \mathscr{L}_\emptyset \rangle$, is defined as $\Sigma_c \triangleright \Sigma = \langle \mathscr{S}_c \times \mathscr{S}, \mathscr{S}_c^0 \times \mathscr{S}^0, \mathscr{I}, \mathscr{O}, \mathscr{R}_c \triangleright \mathscr{R}, \mathscr{L} \rangle$. The transition relation $\mathscr{R}_c \triangleright \mathscr{R}$ is such that Σ_c and Σ either evolve independently by executing internal τ transitions, or evolve concurrently by executing input and output actions.

Due to the asynchronous nature of Web service interactions, and in order to guarantee a correct behavior of the composite service, there is the need to rule out explicitly the cases where the sender is ready to send a message that the receiver is not able to accept. According to [62], a state s is able to accept a message a if there exists some successor s' of s, reachable from s through a (possibly empty) sequence of τ transitions, such that an input transition labeled with a can be performed in s'. This intuition is captured by the notion of τ-closure(s), defining the set of states reachable from s through a chain of τ transitions.

In [62], the composition problem for domain Σ and composition goal ρ consists in generating a STS Σ_c that controls Σ so that its behavior satisfy the requirement ρ (according to a formal notion of requirement satisfaction).

Intuitively, a controller is a *solution* for a requirement ρ if it guarantees that ρ is achieved. That is, if every run σ of the controlled system $\Sigma_c \triangleright \Sigma_{\|}$ ends up in a state where ρ holds.

6.5.3 Application to Services

The approach has been implemented in the ASTRO framework. Similarly to Section 6.4, the basic idea is that existing services can be used to construct the planning domain, composition requirements can be formalized as planning goals, and planning algorithms can be used to generate plans that compose the published services. As outlined above, in ASTRO each service is represented as a state transition system. These can be obtained from abstract WS-BPEL protocols, and thus are easy to come by in practice.

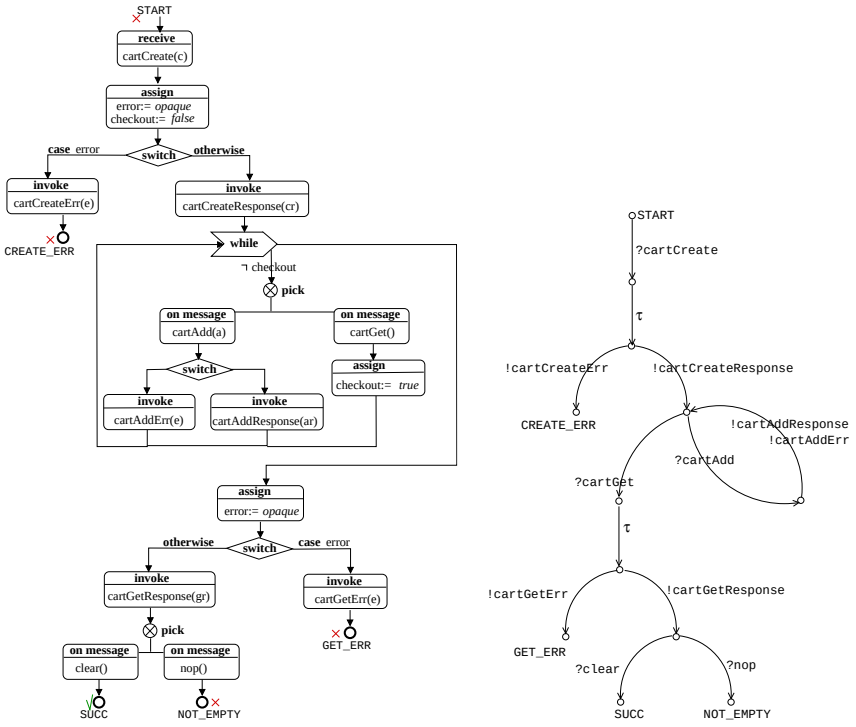
The ASTRO framework has been widely adopted to deal with the different aspects of the Web service composition problem. In particular, ASTRO has been enabled to specify complex data-flow [50] and control-flow composition requirements [11] and an abstraction-based approach for composing services that manipulate complex, infinite-range data domains [61]. The framework has been implemented as a prototype automated composition tool, namely WS-Compose, and integrated in the ASTRO Toolset [7], a toolkit providing an integrated environment for the composition of Web services.

6.5.4 Example

The example we present in this section is taken from the *e-Bookstore* composition scenario [51] where the aim is to automatically synthesize an application that allows to order books through the Amazon E-Commerce Services and buy them via a secure credit card payment transaction offered by Banks of Monte dei Paschi di Siena Group (MPS). This composition scenario is particularly challenging since all component services are real Web services exporting complex interaction protocols and handling structured data in messages.

In particular, we will consider the Amazon Virtual-Cart (AVC) Web service and show its encoding as a STS. For a complete description of the e-Bookstore scenario please refer to [49].

Figure 2(a) represents a compact representation of the abstract WS-BPEL protocol of the Amazon Virtual-Cart (AVC) service. According to this process, once the AVC receives a request to create a new cart and the operation is successful, the client can start to add items and eventually checkout its shopping cart. If the



(a) WS-BPEL specification (b) STS encoding

Fig. 6.2: The Amazon Virtual-Cart Web service.

checkout is successful, the client can either clear the cart or keep its content for future use. In all these interactions if something goes wrong the AVC sends an error message describing the reason of the fault. The final states of the protocol are marked either as successful (symbol $\surd$) or as failing (symbol $\times$) states. These minimal semantic annotations are necessary to distinguish those executions that lead to a successful completion of the interaction from those that are failed. As explained in [51] this information are exploited in the definition of the control-flow composition requirements.

Figure 2(b) presents the STS encoding of the AVC service. Please refer to [12] for the complete description of a translation procedure that allows to encode WS-BPEL processes as state transition systems.

6.5.5 Discussion

The formal framework presented in this Section is primarily motivated by the necessity to provide an automated composition approach that is able to tackle real world Web service composition problems. Among the most important characteristics are (i) the possibility to specify both control and data flow composition requirements, (ii) the ability to consider complex stateful processes as component services, and (iii) the capability to produce an executable, ready to be deployed, composite service. We briefly summarize some of the details involved in tackling these challenges.

In [50] the authors propose to specify requirements on the data flow through a set of constraints that explicitly define the valid routings and manipulations of messages that the new composite service can perform. In particular, data flow composition requirements are defined through an intuitive graphical notation, the *data net*, i.e., a graph where the input/output ports of the existing services are modeled as nodes, the paths in the graph define the possible routes of the messages, and the arcs define basic manipulations of these messages performed by the composed service. Finally, the authors show how to encode these data constraints within the composition domain in an efficient compositional way.

Even though the data net approach allows to specify complex data flow composition requirements, it does not encode data within the composition domain (the states of the domain simply model the evolution of the processes). As a consequence, it is not possible to reason on data when searching for a solution and, in particular, on the conditions ruling the flow of operations and interactions of the component services. To face the data challenge, in [61] the authors propose an abstraction-based approach for handling data, which possibly ranges over an infinite domain, in a finite, symbolic way.

A limitation of the presented approach is the fact that the specification of control-flow and data-flow composition requirements may be time consuming and reduces the applicability of the approach in more dynamic applications (e.g., requiring to substitute/adapt on-the-fly the service components to be used). A significant step in this direction is done by the work in [11] through the usage of business objects that allow to detach the specification of composition requirements from the component service implementations. With respect to the specification of data flow requirements, an idea can be to add semantic annotations to the data used in the component services (similarly to what is done in [5, 4, 45] and then apply semantic matching and reasoning techniques to automatically derive the data links between message parts in order to obtain a first version of the data net diagram that can then be refined by hand.

Another limitation of this approach is that it scales-up well if we assume that we already selected the subset of relevant component services participating to the composition. As shown in [10], the approach described in Section 6.4 can be efficiently used to filter the component services before applying the automated synthesis proposed in here.

6.6 Conclusion

Automation in the handling of Web services requires their annotation with suitable information about their content. Research into this has come up with a broad range of approaches, drawing on several research areas, prominently on Artificial Intelligence and Databases. We have herein introduced four of the most wide-spread formalizations. Description Logics and Logic Programming serve to annotate service input/outputs for better service discovery; Planning for Service Chaining and Planning for Service Interactions require more detailed information about the service behavior, and provide a form of automatic programming composing atomic services to more complex units. All of the approaches involve a form of reasoning and are thus computationally costly in the worst-case; but practical methods can be designed.

Together, the paradigms just described form the basis of service discovery and composition in the Semantic Web Services area. The next chapter gives an overview of that area.

References

1. International health terminology standards development organisation - SNOMED CT. `http://www.ihtsdo.org/snomed-ct/`.
2. A. Ankolekar et al. DAML-S: Web service description for the semantic web. In *ISWC*, 2002.
3. V. Agarwal, G. Chafle, K. Dasgupta, N. Karnik, A. Kumar, S. Mittal, and B. Srivastava. Synthy: A system for end to end composition of web services. *J. Web Semantics*, 3(4), 2005.
4. R. Akkiraju, B. Srivastava, A. Ivan, R. Goodwin, and T. Syeda-Mahmood. Semaplan: Combining planning with semantic matching to achieve web service composition. In *Proc. of IEEE International Conference on Web Services (ICWS'06)*, 2006.
5. J. L. Ambite and D. Kapoor. Argos: a framework for automatically generating data processing workflows. In *Proc. of the 8th annual international conference on Digital government research (dg.o'07)*, 2007.
6. M. Arenas, L. Bertossi, and J. Chomicki. Specifying and Querying Database Repairs using Logic Programs with Exceptions. In *Proc. of the 4th International Conference on Flexible Query Answering Systems*, pages 27–41. Springer, 2000.
7. ASTRO. Project ASTRO: Supporting the Composition of Distributed Business Processes - http://astroproject.org.
8. F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2003.
9. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
10. P. Bertoli, J. Hoffmann, F. Lecue, and M. Pistore. Integrating discovery and automated composition: from semantic requirements to executable code. In *Proceedings of the IEEE 2007 International Conference on Web Services (ICWS'07)*, 2007.
11. P. Bertoli, R. Kazhamiakin, M. Paolucci, M. Pistore, H. Raik, and M. Wagner. Control Flow Requirements for Automated Service Composition. In *Proc. of the IEEE International Conference on Web Services (ICWS09)*, 2009.
12. P. Bertoli, M. Pistore, and P. Traverso. Automated composition of web services via planning in asynchronous domains. *Journal of Artif. Intell.*, 174:316–361, March 2010.

13. S. Biundo, R. Aylett, M. Beetz, D. Borrajo, A. Cesta, T. Grant, L. McCluskey, A. Milani, and G. Verfaillie. PLANET Technological Roadmap on AI Planning and Scheduling. http://planet.dfki.de/service/Resources/Roadmap/Roadmap2.pdf, Dec. 2003.
14. D. Calvanese, G. D. Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The *l-lite* family. *Journal of Automed Reasoning*, 39(3):385–429, 2007.
15. A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning via model checking: A decision procedure for ar. In *Proc. of the 4th European Conference on Planning*, pages 130–142, 1997.
16. A. Cimatti, F. Giunchiglia, E. Giunchiglia, and P. Traverso. Planning as model checking. In *Proc. of ECP*, pages 1–20, 1999.
17. A. Cimatti, M. Roveri, and P. Bertoli. Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, 159(1–2):127–206, 2004.
18. T. O. S. Coalition. OWL-S: Semantic Markup for Web Services, 2003.
19. I. Constantinescu, B. Faltings, and W. Binder. Large scale, type-compatible service composition. In *2nd International Conference on Web Services (ICWS-04)*, pages 506–513, 2004.
20. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Computing Surveys*, 33(3):374–425, 2001.
21. J. de Bruijn and S. Heymans. A semantic framework for language layering in wsml. In M. Marchiori, J. Z. Pan, and C. de Sainte Marie, editors, *Proceedings of the First International Conference on Web Reasoning and Rule Systems (RR 2007)*, volume 4524 of *Lecture Notes in Computer Science*, pages 103–117. Springer, 2007.
22. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The Diagnosis Frontend of the dlv System. *AI Communications*, 12(1-2):99–111, 1999.
23. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLVK system. *Artificial Intelligence*, 144(1-2):157–211, 2003.
24. T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the Semantic Web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.
25. D. Fensel, H. Lausen, A. Polleres, J. de Bruijn, M. Stollberg, D. Roman, and J. Domingue. *Enabling Semantic Web Services– The Web Service Modeling Ontology*. Springer-Verlag, 2006.
26. R. E. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
27. M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J. Artificial Intelligence Research*, 20:61–124, 2003.
28. A. V. Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *JACM*, 38(3):620–650, 1991.
29. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of International Conference on Logic Programming (ICLP 1988)*, pages 1070–1080. MIT Press, 1988.
30. A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.
31. M. Ghallab, D. Nau, and P. Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann/Elsevier, 2004.
32. A. Gonzalez-Ferrer, J. Fernandez-Olivares, and L. Castillo. JABBAH: a java application framework for the translation between business process models and htn. In *Proceedings of the 3rd International Competition on Knowledge Engineering for Planning and Scheduling (ICKEPS'09)*, 2009.
33. B. N. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logic. In *Proc. WWW 2003*, pages 48–57. ACM, 2003.
34. V. Haarslev and R. Moller. Description of the RACER system and its applications. In *Proc. of Description Logics 2001*, 2001.

35. M. Helmert. Concise finite-domain representations for pddl planning tasks. *Artificial Intelligence*, 173(5-6):503–535, 2009.
36. A. Herzig and O. Rifi. Propositional belief base update and minimal change. *Artificial Intelligence*, 115(1):107–138, 1999.
37. J. Hoffmann, P. Bertoli, M. Helmert, and M. Pistore. Message-based web service composition, integrity constraints, and planning under uncertainty: A new connection. *J. Artificial Intelligence Research*, 35:49–117, 2009.
38. J. Hoffmann and S. Edelkamp. The deterministic part of IPC-4: An overview. *J. Artificial Intelligence Research*, 24:519–579, 2005.
39. J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *J. Artificial Intelligence Research*, 14:253–302, 2001.
40. J. Hoffmann, I. Weber, and F. M. Kraft. SAP speaks PDDL. In *Proceedings of the 24th AAAI Conference on Artificial Intelligence (AAAI'10)*, 2010.
41. I. Horrocks. The FaCT system. In *Automated Reasoning with Analytic Tableaux and Related Methods: International Conference Tableaux'98*, number 1397 in LNAI, pages 307–312. Springer, 1998.
42. D. Hull, E. Zolin, A. Bovykin, I. Horrocks, U. Sattler, and R. Stevens. Deciding Semantic Matching of Stateless Services. In *Proceedings, The Twenty-First National Conference on Artificial Intelligence (AAAI) and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*, pages 1319–1324, 2006.
43. M. Krötzsch, S. Rudolph, and P. Hitzler. Description logic rules. In *Proc. ECAI*, pages 80–84. IOS Press, 2008.
44. U. Kuter, E. Sirin, D. Nau, B. Parsia, and J. Hendler. Information gathering during planning for web service composition. *J. Web Semantics*, 3(2-3):183–205, 2005.
45. F. Lecue, A. Delteil, and A. Leger. Applying abduction in semantic web service composition. In *Proc. of IEEE International Conference on Web Services (ICWS'07)*, 2007.
46. N. Leone, P. Rullo, and F. Scarcello. Disjunctive Stable Models: Unfounded sets, Fixpoint Semantics, and Computation. *Information and Computation*, 135(2):69–112, 1997.
47. V. Lifschitz. Answer Set Programming and Plan Generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.
48. Z. Liu, A. Ranganathan, and A. Riabov. A planning approach for message-oriented semantic web service composition. In *22nd National Conference of the American Association for Artificial Intelligence (AAAI'07)*, 2007.
49. A. Marconi and M. Pistore. Synthesis and composition of web services. In *Formal Methods for Web Services*, pages 89–157. Springer Berlin / Heidelberg, 2009.
50. A. Marconi, M. Pistore, and P. Traverso. Specifying Data-Flow Requirements for the Automated Composition of Web Services. In *Proc. of Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*, 2006.
51. A. Marconi, M. Pistore, and P. Traverso. Automated Web Service Composition at Work: the Amazon/MPS Case Study. In *Proc. of IEEE International Conference on Web Services (ICWS'07)*, 2007.
52. D. McDermott et al. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Committee, 1998.
53. S. McIlraith, T. Son, and H. Zeng. Semantic web services. *Intelligent Systems*, 16(2):46–53, April 2001.
54. S. McIlraith and T. C. Son. Adapting Golog for composition of semantic Web services. In *Proc. of the 8th Int. Conf. on Principles and Knowledge Representation and Reasoning (KR-02), Toulouse, France*, 2002.
55. B. Motik and R. Rosati. A faithful integration of description logics with logic programming. In *Proc. IJCAI*, pages 477–482, 2007.
56. B. Motik, U. Sattler, and R. Studer. Query answering for OWL-DL with rules. *Journal of Web Semantics*, 3(1):41–60, July 2005.
57. I. Niemelä and P. Simons. SMODELS - An Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. In *Proc. of the 4th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 1997)*, volume 1265 of *LNAI*, pages 420–429, 1997.

58. OASIS. *Web Services Business Process Execution Language Version 2.0*, Apr. 2007.
59. W. OWL Working Group. *OWL 2 Web Ontology Language: Document Overview*. W3C Recommendation, 27 October 2009. Available at `http://www.w3.org/TR/owl2-overview/`.
60. E. P. Pednault. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 6th International Conference (KR'98)*, pages 324–331, 1998.
61. M. Pistore, A. Marconi, P. Bertoli, and P. Traverso. Automated Composition of Web Services by Planning at the Knowledge Level. In *Proc. IJCAI'05*, 2005.
62. M. Pistore, P. Traverso, and P. Bertoli. Automated Composition of Web Services by Planning in Asynchronous Domains. In *Proc. ICAPS'05*, 2005.
63. M. Pistore, P. Traverso, P. Bertoli, and A. Marconi. Automated synthesis of composite BPEL4WS web services. In *3rd IEEE International Conference on Web Services (ICWS-05)*, 2005.
64. S. Ponnekanti and A. Fox. SWORD: A developer toolkit for web services composition. In *11th International World Wide Web Conference (WWW-02)*, 2002.
65. S. Richter and M. Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artificial Intelligence Research*, 39:127–177, 2010.
66. M. D. Rodriguez-Moreno, D. Borrajo, A. Cesta, and A. Oddi. Integrating planning and scheduling in workflow domains. *Expert Systems Applications*, 33(2):389–406, 2007.
67. R. Rosati. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics*, 3(1):41–60, 2005.
68. R. Rosati. DL+log: Tight integration of description logics and disjunctive datalog. In *Proc. KR*, pages 68–78, 2006.
69. R. Shearer, B. Motik, and I. Horrocks. HermiT: A Highly-Efficient OWL Reasoner. In *Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2008)*, 2008.
70. E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51 – 53, 2007. Software Engineering and the Semantic Web.
71. E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. HTN planning for web service composition using SHOP2. *J. Web Semantics*, 1(4), 2004.
72. T. Soininen and I. Niemelä. Developing a Declarative Rule Language for Applications in Product Configuration. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL 1999)*, number 1551 in LNCS, pages 305–319. Springer, 1999.
73. H. Younes, M. Littman, D. Weissman, and J. Asmuth. The first probabilistic track of the international planning competition. *J. Artificial Intelligence Research*, 24:851–887, 2005.