Chapter 4
Composition Without a Given Structure

Every approach discussed in this section solves some form of a planning problem. A planning problem asks for a sequence of actions that an agent must perform in a problem domain Σ in order to reach a goal situation s^* starting from an initial situation s_0. The problem domain Σ is defined based on a logical language $\mathcal{L}$, which may be propositional logic or (some variant) of first-order logic. It consists of a countable set of states, a countable set of actions that can be performed by the agent, and a state transition function that defines how the agent can move through the state space through actions. The states are described as formulas over $\mathcal{L}$, and the initial state s_0 and goal state s^* belong to the state space.

The analogy between AI planning and automated service composition is as follows. Let us assume that we want to find the *implementation* for a service operation for which we currently have only a signature and logical preconditions and postconditions. The preconditions describe knowledge that may be assumed to be true at time of invocation, and the postconditions say what is true after the (successful) invocation. Then, we can think of this as a planning problem where the initial situation s_0 corresponds to the preconditions, the goal situation s^* to the postcondition, the state space corresponds to the semantic states of the thread that will later execute the implementation, and the actions correspond to invocations of existing service operations.

The service composition problem addressed this way is not generally a *classical* planning problem [105]. Classical planning, which is subject of almost all available planning tools, assumes that the state space of the planning domain is finite. This is often possible even if $\mathcal{L}$ is a first-order logic language by *grounding* the predicates using a finite set of objects that is assumed to exist in the environment. However, in the case of software composition, this set of objects corresponds to the data containers (programming variables) that are used to pass data between operations, and the number of these containers is not bound in general.

F. Mohr, *Automated Software and Service Composition*,
SpringerBriefs in Computer Science, DOI 10.1007/978-3-319-34168-2_4

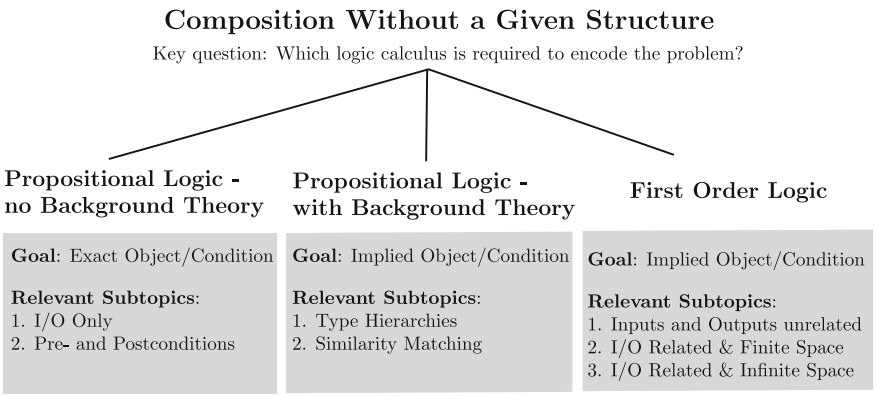

Fig. 4.1 Composition problems where no structure is given

We can identify three subclasses for the approaches within this class based on the underlying logical language $\mathcal{L}$. Figure 4.1 shows an overview over the three subclasses.

1. Many approaches *canonically correspond* to a propositional logical planning problem in that there is exactly one planning action for each service operation with an obvious translation. Section 4.1 discusses these approaches.
2. Many approaches use some sort of (possibly first-order) background knowledge such as type hierarchies that must be encoded in additional planning actions. The transformation to a classical planning problem can be done in linear time. These approaches will be discussed in Sect. 4.2.
3. The third subclass comprises approaches that are based on FOL, which are discussed in Sect. 4.3. Here, operation descriptions may contain predicates with two places or more. Not all of these encodings can be reduced to propositional logic, and if they can, this translation cannot be done in polynomial time.

The main difference between the first two subclasses and the third one is that only approaches in the third subclass allow to relate data to each other and to model the data flow. Approaches discussed in Sects. 4.1 and 4.2 assume that we have only unparametrized knowledge about data; e.g., that an object x *is* a client but not that he is *attended* by some employee y unless y is fixed a priori. Hence, approaches in the third subclass are an order of magnitude more expressive than the approaches in the first two subclasses.

Note that, for better readability, the conclusion of this chapter is found in Chap. 5. The body of the chapter is very long, and I felt that a conclusion of all approaches is better off in the general conclusion. Of course, every section within the chapter is closed with a conclusion in order to summarize the respective subfield; only the general conclusion is found in Chap. 5.

4.1 Propositional Systems Without Background Theory

Approaches of this class assume that service operations are functionally specified either through their inputs and outputs or in terms of propositional preconditions and postconditions. Correspondingly, the goal is either to derive a set of desired outputs from a given set of inputs or to find a composition that guarantees a desired (propositional) postcondition to hold when invoked on a given precondition. The first case is simply a special case of the second one, interpreting the inputs $\{i_1, \ldots, i_m\}$ and outputs $\{o_1, \ldots, o_n\}$ of operations as propositions that are conjunctively connected. Note that inputs and outputs sometimes refer to *names* of the data ports and sometimes to the *types* of data ports, but this difference is irrelevant for the composition process.

Given this type of operation specification, we can create propositional planning actions in linear time. For every service operation o_i, we create exactly one planning action a_i. The precondition of a_i corresponds either to the conjunction of inputs or to the precondition of o_i, depending on the type of operation. Likewise, the postcondition of a_i corresponds either to the conjunction of outputs or to the postcondition of o_i, depending on the type of operation. The state space is defined by the powerset of the set of all propositions induced by inputs, outputs, preconditions, or postconditions occurring in the description of any operation.

The number of approaches presented in this section should not hide the fact that most of them solve problems that are trivial or at least very simple. If we assume that the operators are given in advance—and we are not aware of an approach that does not make this assumption—we can create look-up tables in a preprocessing step that allows us to answer queries in constant time. But even if we do not apply such a preprocessing step, most of the composition problems are still solvable within polynomial runtime. This is simply because every service is contained at most once in a composition. Of course, if operations have negative preconditions or postconditions, or if the goal is to find solutions that optimize QoS-properties, the hardness of the underlying problem increases. However, most approaches considered here do not take these aspects into account and, hence, address extremely simple and practically largely irrelevant problems.

4.1.1 IO-Based Composition

Approaches discussed in this section rely only on the *names* or the *types* of parameters of operations. Service operations are not expected to have semantic annotations in terms of preconditions or postconditions. The planning actions can be defined straight forward. For every service operation, there is one action with preconditions corresponding to the names or types of the inputs and positive postconditions corresponding to the names or types of the outputs. The actions do not have negative postconditions.

4.1.1.1 Forward Search

Thakkar et al. propose a naive forward chaining approach to solve the problem [149]. Given the set of available inputs AI, they iteratively add every service operation to the composition whose inputs are a subset of AI and add the outputs of that operation to AI. The process terminates if all required outputs are contained in AI or if all service operations have been added to the composition. However, the unguided forward chaining implies that the composition also contains service operations that are completely irrelevant for obtaining the desired outputs. Nonfunctional properties are also not considered.

Blake and Cummings add the notion of service level agreements (SLA) to the simple composition algorithm [24]. Considered measurements are up-time (reliability), service rate (execution duration + communication time), maintenance (time that the service must announce its downtime before service is disabled), cost, and renegotiation (time before agreement must be renegotiated). The input of the composition algorithm is a set of provided input parameters, required output parameters, and a vector with bounds for the SLA features. The composition algorithm first performs a forward search in order to identify possible workflows. Every workflow satisfies the SLA bounds of the user and transforms the given inputs into the required outputs. Out of the set of candidates, they then choose the workflow that is best with respect to predefined *priorities* among the SLA measurements. While the technical quality of the approach is rather poor, e.g., the description of the composition routine is significantly flawed, the approach brings some new interesting nonfunctional properties that are not considered by other approaches. However, the discussion of related work is quite insufficient. For example, the difference to Zeng et al. is not only the type of considered nonfunctional properties but that the composition is not based on a template but on a search algorithm. Summarizing, the approach is weak from the functional point of view but provides some interesting nonfunctional properties that are not considered by others.

4.1.1.2 Backward Search

Wu et al. address the same setting as Thakkar et al. but address it through backward chaining using a distance-based heuristic [162]. The basic search algorithm is a backward search algorithm that starts at s^* and prepends operations to the current plan; the state resulting from a prepend step is the old state minus the outputs of the prepended operation plus the inputs of the prepended operation. The algorithm stops when the empty state has been reached. This is a formal flaw, because this is usually impossible, and the algorithm should terminate when a subset of s_0 is reached. The choice of operators to be prepended is driven by a heuristic computed in a preprocessing step. Ironically, the heuristic cannot be computed efficiently, because it already explores the whole search space. Apart from that, it is unclear why a heuristic is needed at all for this unduly simple problem.

Another approach of this section is presented by Matskin et al. [95]. The concrete composition algorithm is not even described, but the requests are of the same type as in the case of [149]. There seems to be no relevant novelty.

Pu et al. perform composition based on complex input and output types [125]. The difference to the above approaches is that inputs and outputs are not only described by atomic parameter names but by complex data types as used in XML schema. Given the lack of semantics, which is already a conceptual shortcoming of the simple problems described above, this approach must be considered almost absurd. The main problem of the syntactical approaches, namely that the human must check the proposed solutions if they are semantically valid, is much worse in this setting, because the semantics faults are harder to track. For example, we could require a type *MyClient(cname,transaction[0,*])* that contains the client name and a list of her transactions based on the two types *Client(cname)*. There are two services: The first receives a *Client* and returns the associated *Employee*. The second accepts *Employee[0,*]* and returns the last *transaction* that was approved by every employee listed in the input. The algorithm provided by Pu et al. finds a solution that determines the employee of the client and creates a list only with this employee. This list is passed to the second service, which delivers a transaction, which is again inserted into a new list of transactions (of length 1). Then, a new complex type is created, together with the client name from the beginning and the list of transactions. Now the resulting type contains a list of transactions, but it contains all the transactions that were approved by the employee associated with the client, which is obviously not what was originally intended. The usage of a cost measure slightly alleviates this problem, but the general problem remains.

4.1.1.3 Dependency Graph-Based Approaches

There are a number of approaches based on the so-called dependency graphs. The idea of dependency graphs is to capture relations among services within a graph structure, which is then used to construct a composition.

Initial Models

First Brogi et al. present an approach that considers ontological matchmaking [29]. The initial situation s_0 and the goal situation s^* are sets of desired ontological concepts, e.g., *username* or *address*. First, their algorithm creates a *dependency graph* (DG) that consists of data nodes N_D and process nodes N_P. Combined with the concepts in s_0 (s^* respectively), the data nodes N_D constitute a set I (O respectively) of usable (required) data. For the creation of the dependency graph, they iteratively run a matching algorithm that identifies services that can either work with the data in I or provide at least one element of O. This matching step considers not only exact matches, but also subsuming types, e.g., services are also selected if they require a more general type than the one available. For every matching service, the algorithm adds one process node for the service (unless it is already inserted) and one data node for every input and output concept of the service that is not part of the graph

already. It then adds an edge from the nodes of the input concepts to the respective process node and edges from the process node to the nodes of its outputs, respectively. This process ends when no more service can be inserted into the dependency graph. Second, it constructs a concrete composition from the dependency graph by first determining the relevant processes using backward chaining and then, out of this set, computing a sequence of "firable" processes; that is, it creates a sequence of services that can be invoked with the given inputs and that obtain the desired outputs.

While there is some novelty in the construction of a dependency graph, the improvement compared to simple backward chaining seems to be rather slim. The novelty of the dependency graph is that it defines a structure in a preprocessing step that then helps avoid decisions during the search process that would yield dead ends. The absence of quality values makes it hard to qualify different solutions, so the qualitative advantage or disadvantage compared to simple backward chaining or forward chaining is not clear. Given the little relevance of the setting due to the lack of semantics, however, there is no gain in going into a more detailed discussion about this point.

Almost at the same time, a similar approach was presented by Hashemian et al. [54, 55]. Instead of having data nodes and process nodes, the dependency graph in their approach has only data nodes, and there is an edge between node v_1 and v_2 if at least one service has v_2 as an output and requires v_1 as an input; the service may also require more inputs and produce more outputs. The edge is labeled with the *set* of services that satisfy this property. The query is defined by a set of pairs of input and output concepts, which are called dependencies; in [55], a query is a pair of *sets* of inputs and outputs, respectively. For each such pair and for every output concept in the pair, the algorithm searches for a path from the inputs to the respective output concept. An additional feature considered in [55] is the cardinality of inputs and outputs; that is, it can be defined that two objects of a particular type are needed instead of only declaring that "some" object of that type is needed.

There are some difficulties with the way how Hashemian et al. make use of the dependency graph. First, the introduction of cardinalities does make sense in this setting. More precisely, it does not matter if a service requires one object of the type *city*, two such objects, or any other constant number. Once it is clear that at least one such object is needed by a service, we need a plan of how to achieve it. But when we have this plan, it can be used arbitrarily often again to produce further objects of that type (which simply yields two equal objects). Second, the composition approach presented by Hashemian et al. is also unsound in that compositions which may contain services whose inputs are not completely provided. For example, if we want to get from concept a to concept b and there is a service that has two inputs a and a' and needs both to produce b. Then their approach will return the service as a solution, because it defines an edge between a and b in the dependency graph. The fact that some other object a' is required may be encoded in the label but is not relevant in the path finding problem; this requirement is simply omitted. Summarizing, their approach is hardly suitable for solving the tackled problem.

A third approach at that time considering something like a dependency graph was presented by Liu et al. [88]. Here, a structure called "deduced network" is computed

in order to determine the possible compositions. The novelty here is that execution prices are considered. However, the quality of this approach is rather poor, because they compute a composition for each output separately and then consider all possible combinations of solutions for the individual subgoals. Among these, they choose the composition with the optimal cost, but it is for example not clear, whether the same service counts twice.

Extended Models

Akkiraju et al. apply an hybrid search to solve the composition problem [5]. The algorithm receives a set of services, a set of provided concepts, and a set of required concepts. First, it computes which services may be relevant for a solution through backward search. Using the remaining services, they then perform a forward search that is guided by a heuristic that is not explained. The approach does basically the same as the algorithm presented by Brogi et al. [29]. Even though it considers ontological concepts in the evaluation of the quality of a solution, the relations among the concepts and their similarities are apparently not considered in the search process itself. Hence, there is no significant novelty in this approach.

Zhou et al. solve the composition problem based on binary trees that encode the dependencies among services [170]. The basis of the computation is a so-called *complete service invocation tree*. Based on this tree, other data structures are derived in order to find a composition. There is no significant novelty in the approach over earlier approaches; in particular, nonfunctional properties are not considered. Since relevant-related work is practically not discussed (in fact, none of the formerly discussed approaches within this section is mentioned), I cannot identify any novelty.

Bouillet et al. describe an approach that solves the same problem as the approaches discussed by Akkiraju et al. [27, 28]. The difference is merely terminological, because they refer to concepts as tags. Even though they claim to use an ontology and to consider subtypes, the planning algorithm used has no native support for this background knowledge, and it is not explained how this knowledge is provided to the planner. Hence, we do not know if and how the type hierarchy can really be considered in the composition process. The discussion of related work (only Lécué et al. and Akkiraju et al. are discussed) does not reveal a significant novelty neither.

Degeler et al. propose an approach that considers the response time of a composition as a nonfunctional criterion [45]. The underlying model is not explicitly called dependency graph, but it has a very similar semantics to the one discussed above. By a simple forward search in the set of possible data flows among services, they determine the minimum response time that any composition has that reaches a particular concept. Then, they apply a backward search individually for each concept to get the "cheapest" composition for the respective concept. The approach brings no novelty and is significantly flawed. First, the approach is not sound, because the backward search does not consider the case that there are no services that can produce required inputs of a service used to provide a goal concept. Second, the model they use assumes that one creates n compositions if n concepts are desired and that these are all executed in parallel. However, this is not a reasonable composition model, in particular given other nonfunctional properties such as price.

Another solution to this problem was proposed by Blanco et al. [25]. They construct a dependency graph quite similar to the one proposed by Brogi et al. [29] but use the notion of Petri nets instead. The innovation is that they consider transactional properties at a risk level as proposed by El Haddad et al. [50]. The difference to El Haddad et al. is that no template is given, but that the composition algorithm tries to find a composition that transforms a set of given input concepts into a set of required output concepts. Some nonfunctional properties (number of service instances, execution time) are considered through constraints (no optimization). As discussed in Sect. 4.1.2, Petri nets are a quite unsuitable model for service composition. They avoid the mentioned problems by not consuming markings from the inputs places when firing transitions, but then one wonders why they use Petri nets at all. Summarizing, the consideration of transactional properties is a novelty, but the model used in the approach is not convincing and the overall problem of finding compositions for concept transformation is still rather irrelevant.

4.1.1.4 Application of GraphPlan

The first one to apply the GraphPlan algorithm [26] to this type of composition problem were Rahmani et al. [126]. The basic idea seems to be that the search process is guided by the distance of the nonfunctional properties to the initial solution. However, the composition algorithm is not described in detail, and, in general, the formalism of the paper is significantly flawed. It is not clear how the nonfunctional properties can be reasonably connected with a heuristic for functionality. Apart from that, given the simplicity of the problem, it is also unclear why a heuristic is needed at all.

Yan et al. proposed a modified version of the standard planner GraphPlan that considers QoS-properties of actions in order to solve the problem [165, 166]. In the modified version, every action node is associated with the cost-properties of the respective action and each proposition node is associated with an optimistic estimate of the costs necessary to produce it. The idea of applying GraphPlan in this setting is somewhat awkward, because the actions do not have negative postconditions, so the heart of GraphPlan, which are the mutexes, are not required. So, approach model does not exploit the strength of GraphPlan but inherits its rather complicated planning process; this even forces them to add a solution reduction algorithm. Moreover, the computation of costs is not reasonable, because they assume the cost for a proposition p to be the cost of the action that produces p plus the maximum cost among the properties within the precondition of the action. But taking the maximum here is not correct, because if the propositions in the preconditions of an action are achieved by several independent operations, only the cost of one of them is considered. Summarizing, Yan et al. add two QoS-properties to the composition model but unnecessarily complicate this actually which is very simple problem.

Recently, Zou et al. have added QoS-properties and preferences to the composition model [171, 172]. The input of the algorithm is a set of service operations, a set of input parameter names, a set of output parameter names, a set of QoS bounds, and

weights for the QoS-properties. A service operation w has a set of input names I_w, output names O_w, and values for the QoS-properties Q_w. The set of query input variable names are the initial state s_0, and the query output variable names are the goal situation s^*. A service operation is applicable in a state s iff $I_w \subseteq s$, and the state resulting from the application is $s' = s \cup O_w$. A solution is a sequence of operations such that the obtained state is a superset of s^*. The QoS-properties are aggregated like in [167], and, among the set of valid solutions, the one that optimizes the weighted QoS-aggregation is chosen.

4.1.2 Composition with Preconditions and Effects

Approaches within this section rely on operations that are described in terms of propositional preconditions and postconditions (maybe in addition to inputs and outputs). Hence, for each operation o, we can simply create a planning action a with the same precondition and postcondition. Except ASTRO, all the approaches are monotonic, which means that operations have only positive postconditions; that is, the postcondition only contains positive literals. In ASTRO, operations are part of state transition systems, so the ability of an operation to be fired must be encoded using state literals, which must be negatable. The four paradigms.

4.1.2.1 Constructive Theorem Proving

In [79], Sven Lämmermann proposed an approach to service composition based on so called *meta-interfaces*. The rough idea of meta-interfaces seems to be that they define functionalities (called axioms) in terms of typed variables or constants. A functionality is encoded in terms of propositional logical preconditions and postconditions. A precondition may contain variable names, logical propositions, and subtasks, which are basically lambda-functions that must be solved first. It is satisfied if each of the mentioned variables are known to have been initialized with a value, if the logical propositions are known to be true, and if the subtask has been resolved. The postcondition may contain a variable name, propositions, or an exception; these may be joint also by a disjunctive operator. Given these meta-interfaces, a set of logical rules can be derived. The query fed to the theorem prover is then as follows: Given the rules obtained from the meta-interfaces and a set of variables that is assumed to be set, can we infer that a particular variable can be set?

The two relevant features that most of the other approaches in this section do not have are subtasks and conditional postconditions. Other approaches in this section describe a service with a set of inputs, a set of outputs, preconditions, and postconditions; preconditions and postconditions are conjunctions of propositional atoms. In contrast, Lämmermann allows for subtasks to contain in the preconditions. A subtask itself is also described in terms of preconditions and postconditions, so it can alternatively be seen as an additional input of the type of a lambda-function. Hence, to

invoke the respective operation it is not necessary to provide an object of a particular type but a function that implements the specified functionality. The second feature is the possibility of disjoint postconditions of operations, which allows for exception handling. This forces the composition algorithm to pursue alternative execution runs of the composition it is creating. The resulting compositions reflect this feature by containing exception handling or conditional statements.

Compared to the enormous formal corpus that is introduced to describe the approach, the overall benefit is rather small. As for all approaches within this section, the semantic power of the queries that can be sent to the system is quite small. How interesting can it be to determine whether or not a particular variable can be set? Of course, if we would impose constraints on the properties of the object that we set to a variable, the issue would be more interesting, but this is never the case. The low semantics are an issue of all the approaches discussed in this section, but most of them are very easy to understand while the description of this solution is very complex and little comprehensive in many aspects. For example, the description of meta-interfaces with the variables, constants, subtasks, and axioms is little comprehensive when compared with the simple IOPE models that underly the other approaches discussed below.

4.1.2.2 Classical Search Algorithms

Kona et al. present and approach for automated service composition that includes propositional log [74, 75]. They provide a naive forward search algorithm that reminds one of the work of Thakkar et al. [149]. The two additional features to Thakkar are conditions and ontological concepts, but none of them is really considered in a convincing way. Conditions are only sets of propositions, so there is actually no relevant difference between inputs and conditions for the algorithm. Second, ontological concepts are mentioned but not used in an appropriate manner. More precisely, the subsumes-relation is used on sets of inputs, for which it is not defined. Also, neither the algorithm nor the examples show the usage of any ontological subsumption reasoning. Apart from this, the forward chaining-specific problem of incorporating useless services is not resolved at all, so the solutions will usually also contain many services that are irrelevant for the respective query. Summarizing, the approach brings no relevant improvement compared to earlier attempts.

For the same setting, Sheshagiri et al. propose a backward chaining algorithm [135]. The critics are the same as for Kona et al. except the use of backward-chaining. Using backward-chaining at least saves Sheshagiri et al. from constructing compositions that contain irrelevant service operations. However, the distinction between inputs and preconditions on one hand, and outputs and postconditions on the other hand is obsolete in this form. So the overall model is quite similar to the ones discussed above and brings no actual novelty.

Agarwal et al. developed a system called Synthy that adds contingency planning and QoS-properties to the above-explained approaches [1, 2]. The algorithm has a logical composition phase, which creates an abstract workflow, and a so-called

physical composition phase, where the abstract workflow is instantiated taking into account the nonfunctional properties. Unfortunately, the logical composition phase is not described sufficiently; they only say that they use limited contingency planning, but it is impossible to figure out how this actually works. The second phase then applies a simplified version of the QoS-optimization model proposed by Zeng et al. [167]. Summarizing, the approach is conceptually relevant due to the integration of planning and QoS, but the formal depth is so low that it is impossible to build upon it.

4.1.2.3 Approaches Based on Resource Models

There are basically two approaches that build on the idea that service composition makes use of resources that are processed. The first is based on Petri nets while the second is based on linear logic. I discuss the two approaches in detail.

Petri Nets

Narayanan et al. were the first to introduce Petri nets to model the consumption and production of data in a service composition [113]. The idea is that the set of all services is encoded as a Petri net, and the task is to find a sequence of transition activations that transforms the initial marking into a goal marking. The Petri net is constructed as follows. For each service operation, there is one transition in the network, and there is one place for every possible assertion over the world (logical atom) and every variable name that is an input or output of a service operation. There is a link between a place and a transition if the assertion or the variable belonging to the place is an input or a precondition of the operation. Likewise, there is a link between a transition and a place if the assertion is an output or an postcondition atom of the operation. The concrete query defines the markings of the network in the beginning. The composition problem is to find a sequence of transition activations such that a given goal marking is reached. Note that, even though the behavior is expressed in situation calculus, it is de facto ground to propositional logic, which is why I discuss it within this section. Later, similar approaches have been proposed by other authors.

Linear Logic

Rao et al. proposed a resource-based approach through the notion of linear logic [77, 127, 128, 129]. A linear logic formula is syntactically similar to propositional logic only that it uses the junctors $\otimes$, $\oplus$, $\multimap$ instead of $\wedge$, $\vee$, $\rightarrow$, respectively. The semantics of $A \otimes B$ is that both resources are available, and $A \oplus B$ means that one of the two is available. $\alpha \multimap \beta$ means that the resources are consumed as described in α and new resources are produced as specified in β. In contrast to propositional logic, a proposition may be contained several times in a conjunction or disjunction in order to express how often the respective information is contained. Services have inputs, which are consumed on execution, and outputs, which are produced after execution. In the descriptions, the functional and nonfunctional parts are separated, but this distinction is not relevant in the formal model or for the solver.

Required but nonconsumed properties must be modeled by being both consumed and produced by a service. So similar to the other approaches discussed in this section, the query defines provided inputs and nonfunctional properties/resources on one hand and desired outputs and demanded nonfunctional properties on the other hand. A (very limited built-in) background theory allows to count the resources available and prevents that more resources than available are used.

Discussion of Resource-Based Composition

In spite (or perhaps because) of the attention they gained in the community, we should make clear that these models are substantially unsuitable for the problem of service composition. While the applied modeling techniques may be interesting in industrial manufacturing systems, digital data, which are the resources of interest here, cannot be considered as consumable units. Once a piece of information is created, it can be used arbitrarily often *without* being consumed; there is simply no need to keep track of the *number* of objects available of a particular type. This is the same objection I already discussed for the approach of Hashemian et al. The only acceptable argument given in [127] is the application of these techniques to nonfunctional properties such as budget; for example, the budget is 20 EUR and every service consumes a certain amount of the budget. However, putting these nonfunctional properties on one level with the functional properties, which are also consumed and against any intuition cannot be used for a second time, yields a quite inappropriate and unnecessarily complex model. Even if it is possible to avoid the consumption semantics in individual cases by declaring every input also as an output, this yields a very unnatural and blown up model.

4.1.2.4 Abduction-Based Service Composition

Okutan et al. propose a composition algorithm based on logical *abduction* [115, 118]. In logical abduction, we assume some knowledge base α and an observation β as given, and we are interested in an explanation γ such that $\alpha \wedge \gamma \models \beta$ holds. In the case of service composition, the formula α is a conjunction of service operation encodings (e.g., in terms of rules) and an initial situation, β encodes what shall be known for the outputs of the composition. The task of the composition algorithm is to find the formula γ, which encodes the *application* of service operations. Intuitively, the service descriptions (α) together with the information how the services are used γ explains how the desired outputs β are obtained. In order to cope with the problem that knowledge is bound to *situations*, Okutan et al. use the event calculus to encode the knowledge and the services.

The idea of modeling the composition task as an abduction problem is intuitive and may be an interesting option, but the approach is still quite preliminary and need substantial improvement in order to be comparable with the other FOL-based approaches discussed below. Even though this is not a general limitation of the abductive approach, it is currently restricted to propositional logical preconditions and postconditions. Only one problem arising from this limitation is that a type-

hierarchical evaluation of parameters is not possible. For example, an object of the type employee cannot be used as an input for a service that requires an object of the type person, even though if employee is a subtype of person. There is a basic support for nonfunctional properties (execution duration, price, reliability, availability), but the model is rather poor. For example, increasing costs increase the score of compositions while they should decrease it, and there is no weighting of the qualities. Summarizing, the abduction-based approach presented in [115] is an interesting initial work but still needs several improvements in order to be on the same level as the FOL-based approaches discussed below.

One significant advantage of the abduction-based approach is that it is directly apt for partial ordered composition. That is, the abductive reasoner does not create a totally ordered composition but only fixes the data flow, which defines a partial order on the service invocations. This property reduces the search space size significantly.

4.1.2.5 The ASTRO Approach

Probably inspired by the Roman model, Traverso, Pistore, and Bertoli developed a composition algorithm that considers services as finite automata [23, 121, 122, 123, 151]. The inputs of the composition algorithm are a finite set of finite state automata, which correspond to the existing services, and a *set* S^* of accepted (and possibly ranked) goal states. In ASTRO, the state of a service is a conjunction of *propositional logical* atoms encoding the values of its variables. The state of the considered system as a whole is defined as the product of states of the services; the initial state s_0 is implicitly defined through the product of initial states of the services. The composition algorithm must construct a *controller* that drives the whole system into any of the goal states of S^* by exchanging messages with the services and, thereby, changing their state and the state of the system as a whole. A particular challenge in this setting is that the automata that model the services are not generally deterministic, so the controller must be able to cope with nondeterministic evolvements of the environment it interacts with.

The two main differences to other approaches within this section are the consideration of constraints on the invocation of service operations and the nondeterminism of those operations. For example, the request for the availability of a product could be true or false; while other approaches subsumes these two responses under a type definition, the ASTRO model considers them on the value level (in form of different response messages). These are important aspects, because both of them impose a significant increase of the computational complexity. In fact, one could model the services of the ASTRO model simply as one planning operation and encode the source and target states in the preconditions and postconditions. For example, suppose that a service has a transition t from state s to states $\{s'_1, \ldots, s'_n\}$, then t is an operation of the service, and we could encode it as a planning action in set theory with preconditions s, positive postconditions $s'_1 \vee \cdots \vee s'_n$, and negative postconditions

s. So we can understand the addressed problem as a nondeterministic variant of the other approaches within this section.

Similar to the discussion on the Roman model, my main objections against this approach is the rather low benefit of automation measured as the ratio between specification effort and achievement of automation. First, the user of the ASTRO framework must specify the goal states *in terms of states of services*. In other words, the user has already resolved the *selection problem* by deciding which services are part of the final composition; no other approach makes this assumption. Note that this also makes the consideration of nonfunctional properties obsolete, which are never a topic within the ASTRO framework. Second, the user must not only solve the selection problem but also know the admissible final states of those services and design the query such that it leaves the system in a consistent state. Third, the data flow is not considered in the automatization process, and the user must specify it in advance; in particular, the user must say which inputs of a service are read from which outputs of which other service. Having the data flow completely encoded this way, the parameters occurring in the operations of the services are fixed, and the communication with the services can be understood as sending and receiving parameter-less *signals*. The remaining problem is to find a *tree* that reflects the possible signals exchanged by the controller and the service community. While this problem may or may not be hard to solve from a computational point of view, the user is certainly faster in simply writing the software than to specify all these details for then having the algorithm automate a tiny part of the task.

Apart from these utility objections, there is also a problem with the soundness of the approach. This can seen best in the latest variant [23], which summarizes the efforts of the earlier attempts. The problem is that the controller may invoke service operations with data that is not available. For example, it may request the *shipper* service for an offer for a package of some size *before* the *producer* service is invoked to determine the size. This is possible, because the requirement definition only defines the partners between the data must flow, but there are no restrictions on the availability.

Huai et al. presented an approach based on the ASTRO model that applies query-based learning to solve the composition problem [61]. Similar to the Eagle language developed by Traverso et al., they use computational tree logic (CTL) to encode the composition problem. However, the approach differs from the above one only in the algorithm that solves the problem, so the major critics discussed above hold likewise.

Summarizing, the ASTRO project enhances the propositional-based composition by *conditional branches* and by a service model that considers *usage restrictions* on service operations, but its utility for the user is little convincing. Of course, the consideration of protocols that limit the way how services are invoked is an important feature. Also, the integration of different possible outcomes of service invocations into the controller is an improvement; in fact, these are equivalent to *if-then-else* constructs. Unfortunately, the way how the user requirements are specified is little convincing, and it is not clear why the user should ever make the effort to provide

all these formal specifications. However, these two aspects are not necessarily tied together. One can envision a framework that takes the underlying service model used in ASTRO but works with a different form of requirement definitions.

4.1.3 Concluding Discussion

Table 4.1 summarizes the approaches discussed so far in this section. It shows whether or not an approach considers nonfunctional properties and whether compositions may contain diverging control flows (if-statements in the control flow). Loops are generally not considered by the approaches in this class. I do not distinguish between the actual information that is encoded (parameters or preconditions and postconditions), because this has no effect on the algorithm. The semantics of these propositions is either "a datum of some type x is available" or "some condition c is true", but actually the first assertion is only a special case of the second.

In spite of the number of approaches in this subclass, the relevance of the problem addressed here is quite small. There are two major concerns about most approaches within this section, which I discuss in the following.

First, the problem is technically so simple that the need to invent a new composition algorithm is quite unclear. Unless nonfunctional properties are considered, the composition task can be simply encoded using PDDL and be solved extremely fast using standard planners, which makes many approaches obsolete [5, 28, 29, 54, 75, 95, 135, 149, 162, 171].

Second, the practical relevance of most of the approaches is very small due to the almost complete absence of semantic information. Except for the ASTRO project that almost specifies the whole solution in advance, the description of desired behavior is highly insufficient. Consider that we have a composition problem where we provide an input *Position* and an output *Telephone Number*. There are numerous possibilities for the semantic connection between the desired telephone number and the position. It could be the phone number of the house closest to the position, the number of the mobile phone that most recently called from that position, the number of a local taxi company, the number of an employee responsible for the respective area around the position, etc. It is highly questionable that the composition algorithm returns a composition that realizes the desired semantic relation.

Probably involuntarily, Hashemian et al. show that the semantics of this composition model becomes quite absurd in the case that operations need more than one input of the same type. For example, they suggest a service operation that computes the distance between two cities, i.e. the operation requires two city objects as inputs. However, the composition algorithm has no reason to provide two *different* cities to that operation, and, in fact, their approach simply copies the solution to get the first city to provide the second city, so the objects will be (always) the same. This example shows in a very illustrative way that these propositional techniques can be hardly considered more than heuristics for FOL composition algorithms discussed above that can compute solutions of a relaxed model fast.

Table 4.1 Overview of approaches without a given structure that rely only on parameter names or propositional conditions

Name	QoS	Alt	Particular Strengths	Particular Weaknesses
Thakkar et al.	○	○	-	compositions contain irrelevant operations
Blake et al.	●	○	consideration of service level agreements	formally flawed
Wu et al.	○	○	-	formally weak
Matskin et al.	○	○	-	poor formal model
Pu et al.	○	○	considers complex types	semantic gaps are even harder to track
Brogi et al.	○	○	good formal model	poor semantics, no ontological matching
Hashemian et al.	○	○	-	poor semantics, unsound algorithm
Liu et al.	◐	○	-	poor semantics, inefficient algorithm
Akkiraju et al.	○	○	clear algorithm description	no novelty
Zhou et al.	○	○	good evaluation	poor formal model, no novelty
Bouillet et al.	○	○	-	reinvents terminology, no novelty
Degeler et al.	◐	○	-	unreasonable assumptions, unsound
Blanco et al.	◐	○	transactional properties	Petri net based model
Rahmani et al.	◐	○	considers user preferences	formally flawed
Yan et al.	◐	○	-	unnecessarily complex, conceptually flawed
Zou et al.	●	○	-	no semantics
Lämmermann	○	●	functions as inputs	unnecessarily complicated model
Kona et al.	○	○	consideration of rudimentary conditions	insufficiently formalized, forward chaining
Sheshagiri et al.	○	○	no irrelevant services in solutions	poor novelty compared to I/O-approaches
Agarwal et al.	●	◐	-	insufficiently formalized
Narayanan et al.	○	◐	-	consumption-based model
Rao et al.	◐	◐	-	consumption-based model
Pistore et al.	○	●	usage constraints, non-determinism	automated task very small
Huai et al.	○	●	usage constraints, non-determinism	automated task very small
Okutan et al.	○	○	uses special calculi (event calculus & abduction)	initial stage, QoS model flawed

QoS = Quality of Service (NF-Properties), *Alt* = Compositions with alternative control flows (if-statements)
This table tries to summarize the partially opposed approaches of this class. The sorting within the lines corresponds to the order in which the approaches were discussed. The *double lines* separate the approaches discussed in different subsections from each other. Literature references can be found in the respective discussions of the approaches

The two positively remarkable properties addressed by some approaches of this subclass are the potential *nondeterminism* of operations and the idea of *partial ordered planning* through abduction. Nondeterminism is considered by Lämmermann [79] (through the notion of exceptions) and in the ASTRO project [23]. That is, the composition algorithm must take into account that the invocation of an operation may have several results, and it must find a solution for each of these outcomes. Abduction is sketched by Okutan et al. [115], which is highly interesting due to the partial ordering of operations. Searching for compositions that are only partially ordered greatly simplifies the search space. However, none of these characteristics compensate the shortcomings of the low semantics imposed by the purely propositional preconditions and postconditions.

4.2 Propositional Systems with Background Theory

The only difference between this subclass and the previously discussed one is that there is some kind of background knowledge that must be encoded in addition to the service operations themselves. The most relevant case is the encoding of a type hierarchy, which is discussed in Sect. 4.2.1. For example, one service determines the price of a product in EUR and another service accepts currency objects as input. Now we have the knowledge that every amount in EUR is also a currency value, hence $EUR(x) \rightarrow Currency(x)$. Hence, we would expect that the second service can be run with the output of the first one. However, approaches discussed above cannot connect these two services based on the type hierarchy information.

Again, since approaches in this section only ever ask for the derivation of *some* object of a given type, we do not need the predicate calculus version of the knowledge base. For example, we can rewrite the above rule simply as $EUR \rightarrow Currency$, meaning that "whenever we have some object of type EUR, we also have an object of type Currency". These propositional rules can then be simply encoded as additional planning actions.

The concept of simple type hierarchies can be generalized by the idea of similarity matching. Instead of saying that the output of an operation o_1 can be used as input of operation o_2 if it is as least as specific as the respectively required input type, a similarity function is used instead to decide whether or not the object can be passed in that way. So similarity matching is somewhat a semantic generalization of the strict type hierarchy. In particular, we could have a similarity measure that takes into account several ontologies and tries to match them based on lexical comparisons. Composition approaches that support this type of background knowledge are discussed in Sect. 4.2.2.

From the complexity viewpoint, approaches using similarity functions are slightly more complex to encode as a propositional logical planning problem. The reason is that the similarity function encodes the rules that are needed to represent the concept compatibilities *implicitly*. Computing the existence of such a rule for every pair of concepts requires quadratic time in the number of concepts. However, this translation can still be considered efficient.

4.2.1 Composition with Type Hierarchies

4.2.1.1 Classical Backward Search

A simple backward greedy search was proposed by Weise et al. [161]. The type hierarchy matching is hidden in the implementation of a predicate called "Promising". Compositions are simply ordered by some heuristic c, where c "combines the size of the set unsatisfied parameters, the composition lengths, the number of satisfied parameters, and the number of known concepts". The simplicity of this solution underlines once more the trivial problem character.

The approaches published by Bartalos and Bieliková are based on a simple backward chaining algorithm [15]. The algorithm is based on a predefined graph that defines which services provide data required by other services. This idea is similar to the dependency graph proposed in [29, 54]. In contrast to some other approaches in this section, it considers the ontological type hierarchy. Later, they published improved variants of their algorithms that can deal with simple first-order logic constraints (cf. Sect. 4.3.2.4).

In the same year, Talantikite et al. propose a backward chaining algorithm [148]. They claim that they improve earlier approaches [10, 11] (cf. Sect. 4.2.2) with better runtime through a precompiled structure they call semantic network. However, neither are these claims supported by evaluation nor is the concept of their semantic network sufficiently innovative to constitute a significant improvement; these networks can be computed efficiently also by other approaches. In contrast, the approaches in [10, 11] actually do consider similarity that exceed mere type systems, while Talantikite et al. only consider exact matches and subsumption matches. The approach also considers some nonfunctional properties (exec-time, resource consumption) by ordering solutions according to a predefined preference function. Hence, the approach provides a composition algorithm and some consideration of nonfunctional properties that we were missing in the work of Constantinescu et al. but it brings no significant improvement compared to existing solutions.

Later, Rodriguez-Mier et al. presented a composition technique that performs a heuristic backward search based on a layered dependency graph [132]. The set of services is partitioned into layers such that L_i contains the services whose inputs can be satisfied by the union of outputs of services contained in L_j with $j < i$. The first and the last layer contain only a dummy service with outputs corresponding to the request inputs and inputs corresponding to the request outputs respectively. Then, an A* algorithm is applied to search backwards for a solution. Every node represents a set of (ontological) types that must still be achieved, and the root node is the set of required outputs. For a node n, there is a successor for each set of services whose joint outputs cover the types described in n; for types not coverable in this way, a dummy service is introduced that has the same type has an input an defers the decision of how to obtain it. A node is a solution if it is empty. The overall description and evaluation of the approach is good, and the used heuristic seems to be admissible. Its only drawback seems to be that nonfunctional properties are not considered at all.

4.2.1.2 Contingency Search

The earliest works that consider ontological type hierarchies in service composition were presented by Constantinescu et al. [41, 42, 43]. In this approach, inputs and outputs of service operations have a type with a domain. An operation is applicable if, for each input, we have a variable whose domain is a subset of the domain of the input variable; i.e., the input must have a value that is accepted for the input variable. In the ontological context, this is often called the *subsumption* relation, but Constantinescu et al. consider also non-ontological types, which is why the applicability is defined this way. While bringing ontological matchmaking to service composition was a conceptual novelty at time of publication, the overall quality of their contribution is rather thin. The formal model is partially unsound, e.g., in the definition of the plugin match for services and query in [42], and the description of composition algorithms is insufficient; in fact, a formal algorithm is only specified in [41], and it is kept very abstract. Also, planning with disjunctive postconditions is far from being trivial; however, this is not discussed at all. Summarizing, Constantinescu et al. presented the fundament for ontological-based service composition but the composition algorithm itself is not convincing.

4.2.1.3 Genetic Programming Solutions

In the same paper as already discussed above, Weise et al. also propose a genetic algorithm to solve the composition problem [161]. In every iteration, the compositions in the pool are mutated by removing the first service with probability σ or to prepend a new promising service with probability $1-\sigma$; a heuristic is used as a fitness function. Little surprisingly, the runtime of this technique (that frequently revokes its own decisions) is much slower than the one of the simple search techniques.

Rodriguez-Mier et al. presented an approach that randomly mutates programs based on a genetic algorithm [133]. The basis of the algorithm is a simple process grammar that defines the language of all admissible programs. The fact that they exchange control structures completely at random (e.g., replace an if-statement with a parallel execution or vice versa) almost surely yields tons of absurd compositions. Of course, these may produce the desired output types, but the resulting compositions must be expected to be quite unintuitive and semantically unsuitable. Summarizing, the approach is a technique to "gamble" for programs, but the degree of randomness of programs together with the low semantic level renders it completely irrelevant.

4.2.1.4 Hybrid Techniques

Dependency Graph Composition

Jiang et al. present an approach that stores the optimal QoS value for each concept in order to optimize the global QoS value of the resulting composition [63, 64]. Through

forward chaining, the algorithm first determines the services that can be executed from the initial situation. Then, it iteratively "triggers" each applicable service and updates the QoS value for each concept that is provided by the respective service. The set of applicable services is extended by the concepts that are outputs of the services already considered. After this procedure, they apply a backward search algorithm to find the best composition with respect to QoS-properties.

Unfortunately, the approach exhibits several significant flaws. Not only is the formal model inconsistent in many parts, also the claim that the algorithm provides globally optimal solutions is false. Suppose for example, that concept a is given and b and c are desired. If there is a service that computes b and c and has cost 3, and there is one service each producing b and c, respectively, with cost 2, then the composition will include the two simple services, because they are the cheapest *local* solutions; however, the costlier service would be better here. Also, the QoS properties are unduly simplified into one single value, and the aggregation of these values remains unclear. Since the comparison to related work is also very thin, there seems to be no significant contribution going along with their approach.

Clustering Approaches

Wagner et al. present an approach based on ontological grouping [159, 160]. The composition algorithm receives a set of services, a type ontology, and a specification of a goal service as input. The algorithm consists of two steps. First, a directed graph is computed where the node set corresponds to the set of services, and there is a link between n_1 and n_2 if the service n_2 *subsumes* the service n_1. Subsumption is defined as follows: Service n_2 subsumes n_1 iff for each input (type) of n_1, n_2 has an equal or more specific input (type), and for each output of n_2, n_1 has an equal or more specific output (type). Intuitively, n_1 can be used whenever n_2 can be used. This process yields a graph with several unconnected node groups; every group has a root, which is called the representative (most general service of the group). Second, a backward chaining algorithm iteratively determines the representative services that contribute to the (remaining) goal and add the corresponding cluster to the plan. If a plan is found that does not have any open inputs anymore, it is marked as a solution. The algorithm checks all possible plans and updates the solution whenever a plan with better "utility" is found; utility here is expressed in terms of reliability and price.

In general, the algorithm leaves a rather weak impression. As so often for approaches in this class, the simplicity of the problem hardly justifies the complicated algorithms. First, the description of the algorithm has several conceptual deficiencies. For example, the algorithm tries *every* possible plan, a strategy that can hardly be considered an improvement for runtime. Second, a composition is a set of *links*, but the algorithm does not at all explain how these links are added to the plan; the complete logic is hidden in an opaque function *computeNextStep*. Third, the approach claims to consider nonfunctional properties, but the QoS model is rather weak. Indeed, the computation of the reliability measure as a "failure among all services within the group" is a good idea, but unfortunately this is the *only* considered property (a formula to compute the price is given but cannot be computed deterministically). Apart from these objections, the service model considers preconditions and

postconditions, but these are completely ignored by the algorithm. Summarizing, the approach brings no significant novelty with earlier approaches in the class.

Ma et al. proposed a further approach based on clustering [91]. Services are clustered based on the outputs they produce. Then the clusters are used to compute a possibly appropriate composition, but the paper fails to make clear how this works. The computation is based on a search in a graph that is not formally defined. Hence, it is not possible to verify the soundness or the evaluation of the approach.

4.2.2 Composition with Similarity Matching

An approach based on backward-chaining for types ground in multiple ontologies was presented by Aversano et al. [11]. Going layer-wise, the approach tries in iteration i to find possible sets of services that produce the concepts desired for layer i, where the first layer corresponds to the goal state s^*. Each such set becomes a new node in the search graph, and the algorithm is recursively applied to it. Ontological types of inputs and outputs may stem from different ontologies. In order to determine whether a service produces a desired output, the matchmaking algorithm considers the type name, properties defined on the concept, and the relation to other concepts through the subclass or superclass relation; hence, the matchmaking is not type hierarchical but rather exact matching among possibly different ontologies. The search process is guided by a node evaluation function that is based on nonfunctional properties. The conceptual explanation of the function is reasonable, but a more formal definition would be desirable. Its major weaknesses are that it does not support hierarchical type recognition for types within one ontology, and that the support for nonfunctional properties is rather rudimentary. However, given that it is one of the first approaches in the class, it makes a significant contribution that exceeds the ones made by some of the succinct approaches.

An approach closely related to the dependency graphs explained in the previous section was proposed by Arpinar et al. [10]. The algorithm receives a set of input concepts and output concepts, and it has the task to return a composition that obtains the desired outputs given the inputs. It first determines the similarity for each pair of (o_y, i_x) where o is an output of service operation y and i is an input of a different service operation x. The similarity is computed by their "ontological distance", but it is not explained in detail what this means. The result can be seen as a graph with nodes corresponding to services and with an edge from node y to node x if y has an output o and x has an input i such that the above condition holds. This graph has two distinct nodes, one for the initial situation and one for the goal situation with edges respectively for the query inputs and outputs. Then, for each input of a service, they compute the "shortest distance" for each input starting from the user input; unfortunately, it is not clear what is meant exactly by distance, but it may be the number of edges. Third, for each service, the shortest distance is computed (maximum among all shortest distances among its inputs). Finally, if the shortest distance for the goal node is not infinite, a solution exists (and it has been computed implicitly) during the former algorithm. Even though the approach provides a formal

model, the actual computation of similarity remains rather vague. Also, it is not clear how the quality of services is considered in the composition process.

In 2006, Lécué and Léger presented a composition algorithm based on so-called casual link matrices [83]. The general idea of casual link matrices is to store information about which outputs of services can be used as inputs for other services. The basis for this matrix are casual links between ontological concepts, which make take values 1 (exact match), $\frac{2}{3}$ (subsumption), $\frac{1}{3}$ (plugin), and 0 if no matching is possible. For every concept that is the input of any service, the matrix has a row and a column. In addition, it has a column for each concept contained in the request. A cell at position i, j contains a set of tuples (x, y) where x may be a service with i among its inputs and having an output o whose similarity to the concept j is greater than 0. In addition, x may be the concept j itself to denote that the concept is known; in this case, y has the value 1. The algorithm *Ra4C* is supposed to find a solution through regression-based search, starting from the desired concepts. Intuitively, it figures out candidate services for the missing goals and, for each candidate, it recursively invokes itself for the inputs of that candidate.

Unfortunately, the technical quality of the approach is very poor. In general, the formal part of the paper is not only very complicated but also exhibits several flaws. For example, the definition of the cells of a casual link matrix is not sound, and the proof of the theorem on composability (which is should rather have the status of a proposition) is technically unsound. However, the most crucial flaws are contained in the composition algorithm itself. While the goals β are always treated as a *set* of concepts, the algorithm seems to treat them only as a single concept. Also, the algorithm simply ignores the tuples in the matrix that are defined on concepts instead of services, which makes one wonder why these were introduced. Another problem is that the algorithm returns a logical formula where the atoms correspond to ontological concepts; it is not clear how a service composition can be constructed from this formula. Summarizing, casual link matrices may introduce an interesting concept for ontology-based service composition, but the *Ra4C* algorithm presented in the paper cannot be considered a suitable solution for the composition problem addressed in the paper.

In [82], Lécué and Delteil build on top of the *Ra4C* algorithm in order to only produce robust compositions. The motivation is that the *Ra4C* algorithm also considers links between services that are only valid due to subsumption match, which is not generally sound. For example, it allows to use a person object where an employee is necessary, given that employee is a subconcept of person. Their approach is based on the idea that it is possible to specify so called *extra description* for the more general concept to cast it down to the more specific one. They suppose that these extra descriptions can be computed automatically, but it is not explained how this can be achieved. Hence, their approach does not constitute a convincing improvement.

Another approach considering similarity was presented by Chifu et al. [39]. The approach is similar to the ones discussed above, and the only innovation worth being mentioned is that outputs that cannot be obtained are added as a required input; this makes the approach a little more robust. This is, however, the only new aspect of

their approach, and none of the related work discussed earlier is mentioned in the paper.

4.2.3 Concluding Discussion

We can briefly summarize these approaches by saying that they consider more technical possibilities of connecting operations but do not resolve the semantic shortcomings discussed in the previous section. That is, the consideration of type hierarchies or similarity functions is a nice additional feature but does not resolve any of the core critics discussed in Sect. 4.1.3. As long as we have no concise description of the *behavior*, which is much more than the types of inputs and outputs even though described through semantic concepts, compositions are mostly unlikely to achieve the desired task.

Note that, apart from this discussion, there could be other types of background knowledge imaginable, but there are no approaches using them. For example, we could imagine knowledge of the form "if service k is used in a composition, then service l may not be used" like applied for the case of template-based composition Sect. 3.2. This type of knowledge could not be directly encoded into the planning problem but would have to remain as a constraint on the meta level. However, I am not aware of any approach that exploits this type of knowledge, and, of course, this would not change anything about the above critics neither (Table 4.2).

4.3 FOL-Based Systems

Approaches of this subclass allow to encode behavior on the level of knowledge about identifiable objects. For example, we may talk about two zip codes and are interested in the distance between the two cities belonging to those zip codes; there will be objects for the zip codes, for the cities, and for the distance, respectively. Forming expressions over objects is enabled by first-order logic (FOL).

I organize the approaches within this class into three further subclasses

1. There are approaches that, similarly to the above techniques, do not relate inputs of operations to their outputs. Still, the behavioral description is more complex, because within the set of inputs and outputs respectively, the objects can be related to each other; that is, preconditions can relate the inputs to each other and postconditions can relate the outputs to each other. Section 4.3.1 discusses approaches of this type.
2. If the postconditions of operations may relate outputs of the operation to inputs, the space of possible compositions is generally *infinite*. Approaches that allow for such postconditions but that make assumptions that avoid infinite search space are discussed in Sect. 4.3.2.

Table 4.2 Overview of approaches without a given structure that rely only on possibly ontological types

Name	TH	Sim	QoS	Alt	Particular Strengths	Particular Weaknesses
Bartalos et al.	●	○	○	○	considers ontological types	very superficial algorithms
Talantikite et al.	●	○	◐	○	composition algorithm presented	little innovation
Rodriguez-Mier et al.	●	○	○	○	good formal models	no similarity measures, no QoS
Constantinescu et al.	●	○	○	○	considers ontological types	formally flawed, no concise algorithm
Jiang et al.	●	○	◐	○	-	flawed formalism, conceptually unsound
Wagner et al.	●	○	◐	○	innovative reliability measure	formally weak, little innovation
Ma et al.	●	○	●	○	-	flawed formalism
Aversano et al.	○	●	◐	○	integration of multiple ontologies	minor formal issues, no hierarchy
Arpinar et al.	●	●	◐	○	integration of multiple ontologies	superficial algorithm description
Lécué et al. (a)	●	◐	○	○	casual link matrices	formally flawed
Chifu et al. (a)	●	◐	○	○	concrete algorithms given	no innovation

TH = Hierarchical Type System, *Sim* = Ontological Similarity Matching, *QoS* = Quality of Service (NF-Properties), *Alt* = Compositions with alternative control flows (if-statements)

● = substantially supported, ◐ = partially supported, ○ = not supported, ⊙ = supported but irrelevant for automation

This table tries to summarize the discussions on the approaches of this class. The sorting within the list corresponds to the order in which the approaches were discussed. The *double lines* separate the approaches discussed in different subsections from each other. Literature references can be found in the respective discussions of the approaches

3. Approaches that consider postconditions that relate inputs and outputs of an operation and that do not limit the potentially infinite search space are discussed in Sect. 4.3.3.

4.3.1 Approaches Without I/O-Relations

The two approaches in this section are similar to the ones discussed in Sects. 4.1 and 4.2 except that preconditions and postconditions can contain relational information referring to the inputs *or* outputs (but not both).

Composition of Relational Concepts

Ambite et al. propose a system for the composition of services where inputs and outputs are relations instead of opaque values [8]. Presumably, the input for the algorithm is a set of services, each of which is described by input and output relations, and a query consisting of a set of input and output relations. I write "presumably", because it is never clearly said what a query is; however, this is the most natural interpretation, which is also shared by Hoffmann et al. [59]. Every relation is factored, which means that it is associated with an ontological concept. The planning algorithm applies partial ordered backward search. It maintains an agenda of concepts that have not been achieved yet. In each step, it identifies a candidate service that has as an output a concept that is equal or more specific than one of the concepts in the agenda. A data link is then added between the inserted service and the service taking the produced output. The innovative point is that the algorithm knows the structure of the relations sent between the services and can perform standard relational algebraic operations such as selection, union, etc., to *synthesize* an input needed by a successor service.

The conceptual aspect of integrating a relational view into service composition is innovative, but the approach only partially delivers on its promises. The actual innovation of the approach is that artificial adapter services can be constructed on the fly in order to translate known relations into desired relations. This is discussed in sufficient detail for the translation obtained by the selection operator of the relational algebra. However, they then claim that they also apply such a mediator algorithms for the operations of projection, union, and join. But it is completely unclear how the presented algorithm translates to these operations; in particular the realization of this mediator for projection and join are far from being straight forward.

Note that there is no semantic relation between inputs and outputs of services. That is, we have a great deal of information about the *structure* of inputs and outputs, but we do not know how the output relations of services relate to the input relations. In this sense, the approach does not provide richer behavioral semantics than the above techniques.

Summarizing, the approach presented by Ambite et al. marks a significant improvement for the application of concept-based composition, but its actual relevance can hardly be judged based on the lack of concise descriptions. On one hand,

it allows for mediator-based composition by providing structural information about ontological concepts. This is significantly more than what is possible with any other approach discussed above. On the other hand, the description of the approach is very imprecise on essential questions; e.g., one misses a concise definition of queries accepted by the composition algorithm. Indeed, the formal parts contained in the paper are (for the most part) sound and comprehensive, but the problem is with the parts that are not described. The reader only gets a rough intuition of the inputs and outputs of the composition algorithm, but since they never make an explicit statement, it is not sufficient to reliably classify the capacities of the approach. In particular, the benefit compared to established relational systems, say Prolog, does not become entirely clear.

Planning with Strict Forward Effects

Hoffmann et al. present a composition algorithm based on (conformant) forward search [59, 60, 136]. The algorithm input is a *set* of possible initial states, a desired goal state, a set of service operations that can be applied, and a simple background theory (ontology). Here, a state is a conjunction of ground literals. A service operation o is applicable in a state s with input values X and output values Y iff the objects X are known in s, the output objects are *not* known in s, and if the preconditions of o interpreted under X are contained in s. The requirement that the outputs Y yet do not exist accounts for the idea that the results of each service invocation are stored in new data containers. Using a forward chaining technique, the algorithm extends the current plan with applicable actions. Every node n in the search space is associated with a formula ϕ_n in conjunctive normal form (CNF) that reflects the postcondition of the composition corresponding to it. A candidate n is a solution to the query iff $\phi_n \models s^*$; that is, if the postcondition guarantees that every literal of the goal situation s^* is true.

The approach makes two simplifying assumptions that dismiss the necessity of belief revision. First, every literal in the postconditions of an operation contains at least one output of the operation. This implies that the application of a service operation can never directly produce knowledge that is inconsistent to the former state; this is because every literal contains a constant (of Y) that was not contained in the previous state. Second, it is required that in every clause of the background theory (which is assumed to be in CNF too) the literals share all of the variables, i.e., the variables occurring in a literal are equal for all the literals in a clause. This makes sure that it is also not possible to combine the newly obtained knowledge with the background theory to infer new knowledge that talks only about constants that were already known previously; otherwise, the newly obtained knowledge could yield an implicit contradiction. Problems that satisfy these two properties are said to exhibit *forward effects*.

In order to reduce the number of actions that must be considered, they make another quite serious simplification, which they call *strict* forward effects. The second of the above two conditions is restricted more by requiring that *all* variables occurring in the postconditions of an operation are outputs. The serious consequence of this restriction is that the outputs cannot be related to inputs of the service anymore.

I think that this is assumption is *too* restrictive, but at least this issue is discussed honestly, and the authors also point out that there are still realistic problems that can be solved under this restriction. Under the assumption of strict forward effects, the composition problem is only slightly more expressive than in the propositional systems.

Even though the algorithm uses an (admissible) heuristic, it is highly questionable whether forward search is a good approach for service composition. The key problem of forward search is that it also considers actions that are not relevant for the goal. This problem increases by an order of magnitude in the service composition scenario in which every action creates new objects and, thereby, enables many new actions. In particular, the number of children of each node in the search space increases with each step. It is hardly imaginable that we can get a heuristic that is sufficiently well informed to efficiently guide a best-first search process. Probably, the only hope is to try some hill-climbing strategy and to cut irrelevant elements later on. Of course, in the case of strict forward effects, this problem is relieved by the fact that each operation must be considered at most once. However, in the general case, forward search is probably a borderline hopeless project; their results exhibits enormous search runtimes even for the highly restricted case of strict forward effects.

4.3.2 I/O-Relational Approaches for Finite Spaces

The following approaches describe the behavior of operations by relating the produced outputs to the inputs. The potential infiniteness of the set of compositions is avoided in several ways.

1. The simplest way to make the search space finite is to allow only for compositions that contain an operation at most once.
2. The information integration approach is bound by the fact that all operations work on a *central* data model, which is finite.
3. The approaches applying PDDL bound the model by assuming only a finite number of containers that can be used to pass information among operations.
4. Finally, the technique proposed by Bartalos assumes that the precondition of an operation must completely be satisfied by the preceding operation in the composition, which also bound the set of possible compositions.

4.3.2.1 Limitation of Operation Usage

An approach that makes use of SMT solvers to address the composition problem was presented by Gulwani et al. [53]. Here, the input of the composition algorithm are an input vector, a desired postcondition, and a set of available operations; it is assumed that the composition produces exactly one output whose relation to the inputs is described in the postcondition. Every operation is likewise described by an input

vector and its postconditions. The algorithm creates a composition that is a *sequence* of *all* of the available operations. So the composition algorithm (i) determines a permutation of the operations and (ii) fixes the data flow between them. To this end, they introduce so called *local variables* that reflect the position of an operation in the final composition; since every operation has one output, this index also refers to a datum produced by the respective operation. The algorithm then encodes the integrity constraints on the data flow in a formula and passes it to an SMT solver together with the operation descriptions and the desired postcondition. If the solver finds a data flow such that the desired specification must necessarily be satisfied, the respective data flow, (which imposes also the control flow) is returned.

The limitation that it creates compositions that make use of every operation exactly once is a quite strong shortcoming. The authors argue that unnecessary parts can be "easily" stripped away afterward and that operations that are required several times can be cloned by the user in advance. But neither is this stripping process of "dead code" (which is not dead, since the composition does not contain if-statements) explained in detail, nor is the drawback discussed that arises when the user must know in advance how often every operation is used; why would he then make use of automated composition techniques? Intuitively, the algorithm either considers too many or too few operations.

Apart from this conceptual flaw, the approach exhibits the same problems as the one by Srivastava [143] discussed in Sect. 3.2 even though it avoids some complexity issues due to the restriction to sequential compositions. The implementation and description of operations are not separated from each other, which imposes the same inflexibility as in the case of [143]. The composition process also relies on an SMT solver, but the fact that the solver does not need to guess statements or guards, the complexity is significantly less than in the case of [143]. The obvious consequence is that the potentially achievable programs are much simpler.

One advantage of both this approaches over most service composition approaches is that it allows for rather complex preconditions and postconditions. In general, it seems possible that it works with preconditions and postconditions that are not only conjunctions but arbitrarily structured formulas. For example, the postconditions of the query consist of a conjunction of rules. In contrast, current approaches for automated service composition only allow conjunctions of ground literals.[1]

To summarize, there are relevant recent approaches to program synthesis that exhibit both a significant intersection and significant differences with automated service composition. The most important commonalities are the goal to automatically synthesize software and that this is done on the basis of implicit goal descriptions and with a library of components described through preconditions and postconditions (of unequal complexity). The most important differences are that program synthesis approaches do not distinguish between the implementation and the description of operations, which reduces these approaches to work with very simple operations, mostly numeric or set theoretic ones. Certainly, the fields can learn a lot from each

[1] One exception is [59] where there may several initial states. Also, most approaches interpret the output variables of the request as (implicitly) existentially quantified.

other, and it would be interesting to combine them in the long term view in order to unify the power of both domain theories and interface-based composition.

4.3.2.2 Information Integration

In 2002, an approach related to information integration was proposed by Ponnekanti and Fox [124]. The basis of the approach is an entity structure like an entity relationship model. A query sent to the composition algorithm consists of the entities involved, provided attributes of these entities, constraints on the entities, and the requested attributes or relations for the entities. For example, a query may ask for a composition that works on two objects X and Y of the type *Person*, for both of which the first name and the last name are given as inputs, and for which we are interested in a shortest path to get from the house of X to the house of Y. That is, there is a relation $DrivingDirections(\cdot, \cdot)$ that we want to compute for the pair (X, Y). The controller is assumed to have a table of each attribute and each relation available, which is partially computed by the invocation of services. The data flow between services is fixed in their description that matches the names of these tables maintained by the composed algorithm.

Within its limited range, this technique is substantially better than many of the propositional logical approaches discussed above that ignore the data flow. The advantage is that the communication with services always happens with respect to particular objects, and it is also possible to request the same attribute for two different objects of the same type; in the propositional logic systems, this query type does not make sense. Of course, there are some limitations. For example, one can determine the price of a product as the attribute of the respective product entity, but the price cannot be converted into a different currency. The reason is that the predicates are only defined over entities but not over attributes, and a particular piece of information can only be either an entity or an attribute. Apart from this limitation, the approach is fairly easy to understand and seems to have the potential of reasonable usage in practice.

4.3.2.3 PDDL-Based Approaches

Initial Model

Joachim Peer proposed a technique that composes constant-based service invocations [119]. The algorithm receives a set of services with preconditions and postconditions and a goal specification. As an example, he proposes a goal that requires the composition to "send the name of the city with ZIP code 30313 to the email address john@some.com". Services can be information gathering, e.g., a service that computes the state and the city given a ZIP code, and world-altering services, e.g., a service that sends an email. The composition algorithm consists of two parts. First, a simplified problem is reduced in which constraints on concrete values are ignored.

Then, the information-gathering services of the plan are invoked in order to extend the knowledge about the world. Second, the gathered knowledge is added to the initial situation s_0, and the problem is solved again. The composition problem is encoded in the planning language PDDL, so that it can be solved with any standard planner. The objects encoded in the PDDL problem are constants referring to objects in the real world. Since the objects do refer to concrete data items instead of generic data objects (what would be called a variable in a programming language), the algorithm does not create a composition with a data flow between service operations. This is the same as programming a sequence of function calls where every argument passed to a function call is a *constant* and not an output of previous function calls. These constants are either given initially or obtained through the first phase of the algorithm.

In the presented form, the approach exhibits two major flaws. First, it simply merges input and outputs to the general concept of *parameters* in PDDL. The conceptual problem is that we cannot encode information-gathering service operations in classical PDDL, because then an invocation is not possible unless we already know the desired information, which is simply a normal parameters such as the inputs, in advance. A more detailed discussion of this problem was published earlier by McDermott [97], which is even cited by Peer; however, this issue was simply ignored. Second, the actually interesting part of the algorithm, which is the first phase, is not described. The second phase is simple and could be also considered as a simple Prolog query. The world-altering services, which have no outputs, are encoded as rules, and the knowledge initially given or gathered in the first phase are assertions. But the interesting question is obviously the first phase of the algorithm, in which it is determined for which predicates a partial grounding is desired and queried. Given the fact that this first phase would be the actual contribution, but that it is not discussed at all, the approach does not exceed a preliminary conceptual level.

Extended Models

Klusch et al. propose a PDDL encoding that avoids these problems [71, 72]. The idea is to introduce a special predicate *agentHasKnowledgeAbout*(x) to assert that the object x is available. For each input of a service operation, the predicate is part of its precondition, and for each output, it is part of the postconditions. Having this meta predicate at hand, the planner can only use data objects as inputs that have been made available either in the request or by previous service calls. The approach is based on both HTN planning and classical planning. It first tries to find a solution using a simplified form of HTN planning, and, if no solution can be found that way, it applies a classical planner.

In spite of the generally good idea, there are quite some problems with their approach. First, the overall explanation of the approach is unduly superficial. For example, HTN planning and classical planning are quite distinct approaches, but they simply mix the two without a detailed explanation of how this is done. Second, there are several conceptual flaws with respect to the planning problem definition. For example, the paper uses real world entities in the planning problem, e.g., the patient *Mikka*. But this does not make sense in combination with the *agentHasKnowledgeAbout* predicate, because either we know that Mikka exists (then we can use it) or we

do not (then we cannot even model this object). The problem is that the semantics of the *agentHasKnowledgeAbout* predicate is that it asserts whether or not a *data container* (in programming languages we would call it a variable) has a value assigned or not; hence, it implements the check $x \neq \mathit{undefined}$. However, this semantics does not make sense when applied to real entities. Third, the resulting encoding into PDDL suggests that it can be solved with standard planners, but it effectively cannot due to complexity issues. The reason is that the set of objects in the PDDL problem is the set of data containers that is used to pass information among the service operations, and we do not know in advance how many such containers are necessary. Even for relatively small sizes, e.g., 30, the resulting planning problem cannot be solved even with highly advanced planning tools.[2] In their implementation, they only use one or at most two variables *per type*, which is equal to the assumption that we already know in advance what data we will need; but then, data flow planning is obsolete. Summarizing, the approach brings a small conceptual improvement, but its overall quality is rather weak.

A third approach that is based on a PDDL encoding was proposed by Vuković et al. [157]. The core idea is pretty similar to those of Peer and Klusch et al. The main difference is that no particular predicate for the availability of data is used, such as the *agentHasKnowledgeAbout* predicate in [72]; this makes one wonder how it is avoided that undefined variables are used. The approach lacks from the same complexity problem as Klusch et al. does, even though their evaluation suggests that the approach is efficient. Since none of the earlier approaches [72, 97, 119] is discussed, I cannot identify a particular novelty of the approach.

4.3.2.4 Limitation by Requiring Full Precondition Coverage

Another approach-based based on simple first-order logical preconditions and post-conditions was proposed by Bartalos and Bieliková [14, 16]. In this approach, a service is described by ontologically typed inputs and outputs and by so-called *conditions*. A condition is a formula that contains symbols for predicates, conjunction, disjunction, and negation; so no function symbols or quantifiers are allowed. A composition is a DAG where every node is a service invocation and a link between service s_1 and s_2 exists only if the postcondition of s_1 implies the precondition of s_2. The paper defines the logical implication in an optimistic way, such that condition c_2 is said to be implied by condition c_1 if there is one clause in the disjunctive normal form (DNF) of c_1 that implies at least one clause of the DNF of c_2. A composition is a solution for the request, if, for each desired output, there is one service that provides it. In addition to the explicit conditions, the approach also considers ontological matchmaking in the data flow; outputs can be used whenever they are more specific than what was requested.

The strong restrictions used in the definition of a composition help create a highly efficient composition algorithm but are equally highly limiting. On one hand, the

[2] I used the FastDownward algorithm to verify this claim.

requirement that a service covers the *complete* preconditions of its successor in a composition allows for a preprocessing step in which all possible ways to chain two services can be computed. This allows to answer queries in fractions of seconds. On the other hand, the set of possible compositions is extremely reduced by this assumption, because preconditions of services cannot be composed from two independent operations. For example, consider that we want to use a service that sends some information to all reliable clients that have completed an order in the last month, and suppose that there are two services that compute from a given set of clients all those that are reliable or completed a purchase in the last month respectively. A valid composition invokes one of them with the input set and then the other with the result of the first operation; the result can then be passed to the third processing service. However, this is not possible here, because the preconditions of the third service cannot be satisfied by any of the former two alone. Another issue is that the simplified treatment of disjunctive conditions cannot be considered sound. Summarizing, the approach allows to consider a significant extent of semantics in the service descriptions. An efficient composition of these services is enabled by a simplified evaluation of the conditions and by a rigorous restriction on possible compositions.

4.3.3 I/O-Relational Approaches for Infinite Spaces

Approaches belonging to this subclass consider the possibility of producing arbitrary new information by the application of operations. The invocation of an operation produces (if it has any outputs) a new datum, which can possibly be used as inputs for other operations. The set of possible compositions is infinite, because we can potentially create ever new pieces of information.

4.3.3.1 Term-Algebraic Program Synthesis

The first solutions for automated software composition at all were proposed by Manna and Waldinger [93, 94]. Their approach is based on an algebraic *term transformation system*. The request consists of a precondition and a goal term that shall be computed. The basis for the composition process are transformation rules that assert admissible ways to rewrite terms. For example, an transformation rule $v \cdot 0 \Rightarrow 0$ asserts that one can renounce a factor multiplied with 0. Based on the resolution calculus, they propose a method that allows to rewrite the initially desired goal term into other goals until the trivial goal *true* is reached. The program is obtained by the term unifications used to apply the transformation rules.

The main difference between this type of automatic programming and service composition is that operations are described in terms of other operations. The semantics of an operation in deductive synthesis is encoded in transformation rules. The left-hand side of the rule states the invocation of an operation and the right-hand side states what we know about the result of the invocation; that is, how we can replace

the invocation. For example, the rule $reverse(u) \Rightarrow reverse(tail(u)) <> [head(u)]$ defines the postcondition of inverting a nonempty list u. So the semantics of *reverse* is expressed in terms of itself (recursion) and other operations *tail* and *head*. Rules may also be bound to some condition, which we would call precondition. In a way, the transformation rules have similarities with methods in HTN planning (discussed in Sect. 3.3), because they describe how a term (possibly a complex service) can be rewritten.

When discussing their approach, it is important to distinguish the underlying algebraic calculus from the way how they apply it. My assertion is that the way how they encode composition problems and how they perform deductive synthesis is apparently different from the way how composition problems are encoded today. However, I do not want to give the impression that the algebraic calculus used by Manna and Waldinger is unsuitable for service composition in general. In contrast, it seems that the term transformation system is so general that it could also be used to encode the type of service composition we are using in the planning context nowadays. Still, we can only discuss an approach to the extent to which the calculus is explicitly used for the particular problem; otherwise we could also argue that Turing presented a mechanism that can be used for service composition by proposing a model of computation.

The most crucial problem with deductive program synthesis for today's research is that we are left with the lack of evaluation. Except the very vague explanations in [114], there is virtually no information about the runtime performance of their algorithm on the machines that were recent in the respective time and much less of how those algorithm would perform today. Of course, complexity issues cannot be resolved with (polynomially) faster computations, but at least it would be easier to compare the approaches. Unless somebody reanimates this algebraic approach, deductive program synthesis stands behind service composition like a shadow of which it is unclear how it relates to the currently developed techniques.

Summarizing, while deductive synthesis in the presented form is hardly compatible with a modern view on software development, we can still learn a lot from this early attempt. Of course, the encoding chosen in [93, 94] exhibits a connection between description and implementation that can be hardly considered timely. On the other hand, current composition approaches completely lack built-in operations for basic data structures. It would be advantageous to compose not only business service operations but also set operations such as *head*. Together with the knowledge $y = head(x) \wedge sortedBy(x, price) \Rightarrow cheapestOf(y, x)$, the composition algorithm could be enhanced with very useful theories that help treat different data structures or basic arithmetical operations. Hence, we should rather seek to complement the modern approaches with the early stage attempts.

4.3.3.2 PDDL-Modification

In one of the first approaches so automated service composition, Drew McDermott extended the PDDL specification in order to make it suitable for service composition

[97]. McDermott realized that PDDL lacks the possibility to specify the creation of new information; so he added the notion of step-values, which are like the (single) output of an action. The output values have a type and may or may not have a default value. If an output has a default value and if another service is used whose precondition make assertions about that value, the planner inserts a special predicate *verify* that signals that, in case that a solution is found, a case distinction must be inserted. In an initial run, the algorithm assumes that the verify-predicates are all true. If a solution is found, the algorithm is restarted with the initial situation being the first situation in which a verify-predicate occurs, modified in a way that the statement to be verified is negated. Starting from there, the algorithm tries to find a solution for the alternative branch. In this way, the algorithm is able to compose programs with conditional branches.

Even though the approach does not exhibit particular shortcomings, it has never been adopted or served as a basis to build upon by later approaches. I already discussed some approaches based on PDDL that do not make use of McDermott's modifications. One problem could be that the supposed advantage of PDDL is that it serves as an input for standard planners but that a significant part of the specification is not covered by any planner; this becomes obviously even worse with the additional extension made in [97]. At time of writing, at least the planner Optop written by McDermott himself is available at his website. Bertoli et al. claim that the approach cannot cope with protocol specifications [23], but given the fact that protocols can be encoded simply through propositional assertions in the preconditions and postconditions of services, this claim cannot be justified.

I think that there are three arguments why Optop is not the end of the story for service composition. First, we have seen that nonfunctional properties are an important aspect of service composition, but these are not considered at all. Since there is no straight-forward way in PDDL to consider these properties, another extension of PDDL would be required. Second, a lot of research related to service composition is concerned not only to how to *model* the composition problem but also of how the space of possible compositions is *traversed*. McDermott proposes a search based on a regression-match graph, but there are many other possibilities about how the search space can be traversed. Third, the paper reflects only a preliminary stage of research without any evaluation. We have no information of how the approach performs in comparison to others; the goal and the achievements of the paper is only to give a proof of concept that estimated regression works for service composition. Also, it does not provide for loops, which are inevitable for most applications. Hence, we have seen a sound but rudimentary solution for automated service composition, and there is plenty space for improvements.

4.3.3.3 General Unbounded Search

In our recent works, we have proposed a technique to search for service compositions without a limitation of the number of variables [109]. The input of the composition algorithm is precondition and postcondition as conjunctions of literals, a set of ser-

vices described in the same way, and a vector of bounds for the nonfunctional properties. The algorithm searches backwards starting from the desired postconditions and builds a composition by *prepending* an operation invocation to the current composition in each step. Hence, compositions computed by this approach are only *sequences* of operation invocations. A service operation is a candidate for being prepended if its postcondition contains at least one literal that is required for the precondition of the currently considered composition. During the composition process, the algorithm may introduce an arbitrary number of new variables (as yet undefined sources of some of the inputs of prepended operation invocations). Every (partial) composition is associated with a vector of nonfunctional properties, which are assumed to increase or decrease monotonically. The algorithm returns a *stream* of Pareto optimal compositions.

The algorithm can also insert more complex control structures if these are hidden in building blocks derived from domain independent templates [107, 108]. These templates are more specific with respect to the control flow elements than the ones used by Srivastava et al. in [143], e.g., the rough code within a loop body is already set. This structural restriction increases the feasibility of the approach, because otherwise there would be too many candidate implementations. The templates contain placeholders for boolean expressions (usually of if-statements), service invocations, and auxiliary predicates. For example, a template FILTER takes a set A as input and computes the subset of elements that satisfy a particular property. For every $a \in A$, a (still undetermined) service s is invoked and determines the value of some (still undetermined) property of a. The obtained value is tested against some (still undetermined) condition. The item a is added to the output set A' if this test has a positive result. This template can then be used to, say, filter a set of books by those that are available. In [108], we present a template instantiation technique that can be directly integrated with the composition algorithm described above. Similar to the approach of Srivastava et al. template here are a *possible guide* to find a solution but they do not encode the actual behavior of the composition.

The formal model underlying our approach is almost the same as the one of Hoffmann et al. [59] with the crucial difference that we do not assume strict forward effects. That is, we allow the postconditions of services and the postconditions specified in the query to related outputs to the inputs. On one hand, this difference has a significant computantional impact in that it precludes the possibility to ground the problem to a (finite) propositional model. In fact, the composition problem probably becomes undecidable by this assumption. On the other hand, it allows to specify much richer requirement definitions that are much closer to the intuition of a specification of actual behavior than lose properties of the ingoing and outgoing data. While undecidability is certainly not a desirable theoretical feature, for practical applications of service composition undecidability is not much worse than highly exponential complexities. That is, it does not matter whether the algorithm runs forever or whether it terminates after some weeks or even years saying that no answer exists; either it finds a solution fast or we must implement the desired component by ourselves. So, independently from decidability questions, the goal must be to find solutions fast if they exist; proving that no solution exists is of minor practical importance.

The main drawback of the approach is that it does not support diverging control flow branches. That is, if-statements are only allowed in templates and only if the template postconditions are still deterministic (purely conjunctive); that is, the composition algorithm does not need to plan two or more possible program states in parallel. McDermott resolved this problem by first planning optimistically and then planning the alternative branches afterward. However, the current version of our composition algorithm does not provide this clearly desirable functionality. Hence, the treatment of alternative and diverging control flow branches remains important and, given a backward search, nontrivial future work.

Summarizing, our approach provides an alternative to the model proposed by McDermott that considers nonfunctional properties and complex predefined control structures but that still lacks the ability to compose diverging control flows. Instead of investigating the problem on a language-specific level like PDDL, we prefer an independent mathematical model that defines a search space that can then be traversed by search algorithms such as A^*.

4.3.4 Concluding Discussion

The techniques discussed in Sect. 4.3.1 provide good formal models and interesting ideas, but the lack of relations between inputs and outputs of operations hinders semantically meaningful composition. The more detailed description of inputs and outputs based on first-order logical formulas is an improvement over the approaches discussed in Sects. 4.1 and 4.2, but meaningful composition requires to relate inputs to outputs, which is not the case in these systems. Still, at least for the system proposed by Hoffmann et al. [59, 60], the model has a native support for meaningful composition if the assumption of strict forward effects is dropped. On the other hand, dropping this assumption probably renders the task unsolvable with current planning algorithms, so practically solving the problem without strict forward effects certainly entails quite some work. Another issue of the two techniques discussed here is that they do not incorporate any notion of nonfunctional properties (Table 4.3).

Program synthesis is an interesting field but is somehow out of phase with respect to the underlying operation model. The fact that we can only use operations whose implementation corresponds to the description seems to be a strong limitation. The reader always has the impression that those techniques only work on very specific domains, e.g., numerics and sets. On the other hand, it is quite possible that these composition models can be extended such that they decouple implementation and description. The advantage of such a system would be tremendous because it would allow for operation descriptions with both uninterpreted and interpreted predicates. However, this integration is currently not visible.

The remaining approaches address what I would call the core of automated service composition. The user can specify a desired behavior of the composition in terms of uninterpreted predicates that relate the requested outputs to the provided inputs. Unfortunately, the approaches described by Peer [119], Klusch et al. [72],

Table 4.3 Overview of approaches without a given structure that rely on first-order logic descriptions

Name	TH	QoS	Alt	Loops	Particular Strengths	Particular Weaknesses
Ambite et al.	●	○	○	○	good formalism, structured data	no description of composition queries
Hoffmann et al.	●	○	○	○	great formalism, extendible unbound search	no relation between inputs and outputs
Gulwani et al.	○	○	○	○	good performance, complex postconditions	implem.=descr., permutated operations
Ponnekanti et al.	○	○	○	○	simple algorithm	very limited composition possibilities
Peer	●	○	○	○		outputs are ordinary PDDL parameters
Klusch et al.	●	○	○	○		poor performance through grounding
Vuković et al.	●	○	○	○		no discussion of related work, no novelty
Bartalos et al.	●	○	○	○		
Manna et al.	○	○	●	●	good formalism, recursion	no QoS, no evaluation
McDermott	●	○	●	○	unbounded model, alternative branches	not further elaborated
Mohr et al.	●	●	◐	◐	unbounded model, QoS considered, basic loops	no native support for alternatives

TH = Hierarchical Type System, *QoS* = Quality of Service (NF-Properties), *Alt* = Compositions with alternative control flows (if-statements), *Loops* = Compositions with loops

● = substantially supported, ◐ = partially supported, ○ = not supported, ⊙ = supported but irrelevant for automation

This table tries to summarize the discussions on the approaches of this class. The sorting within the list corresponds to the order in which the approaches were discussed. The *double lines* separate the approaches discussed in different subsections from each other. Literature references can be found in the respective discussions of the approaches

and Vuković [157] exhibit significant formal flaws or are not sufficiently elaborated to actually apply them. Also, nonfunctional aspects play virtually no role. The technique presented by Ponnekanti and Fox [124] is well elaborated and appears quite useful. Probably the only concern against their technique is that the set of possible queries is very limited, because we can only ask for attributes of entities. Among all approaches discussed so far, McDermott [97] and our own work [106] are the only solutions to the composition problem that do not exhibit any of these limitations.

However, even the composition algorithms discussed in [97, 109] can only be considered initial steps. One problem is that none of the two has a complete support for all of the functional and nonfunctional aspects that would be relevant for composition. For example, McDermott does not treat nonfunctional properties or loops. Our own algorithm covers nonfunctional aspects but has only quite limited support for conditional statements and loops. In particular, the consideration of diverging control flows is not possible in a straight-forward manner.

Clearly the greatest challenge of unbounded service composition is to optimize the runtime of the search process *without simplifying the model*. Semantically meaningful composition may induce semi-decidability, which seems to be the case in [97, 109]. However, the strategy should not be to downgrade the problem but to enable a fast finding of solutions if they exist. In practice, the difference between, say, NEXPTIME and undecidability is almost irrelevant, because we do not need a *proof* for the nonexistence of a solution. Either we can find a solution fast or we implement it manually, but we do not care about whether no solution was returned in time because no one exists or because the algorithm was not fast enough to find it. I do not say that it is not a good idea to solve a simplified version of the model in an interior routine whose parameters are iteratively adjusted to continuously expand the search space. But we should not unnecessarily simplify the task itself only to obtain fast algorithms (that solve irrelevant problems).