

Automated Service Composition using Heuristic Search^{*}

Harald Meyer and Mathias Weske

Hasso-Plattner-Institute for IT-Systems-Engineering at the University of Potsdam
Prof.-Dr.-Helmert-Strasse 2-3, 14482 Potsdam, Germany
{harald.meyer|mathias.weske}@hpi.uni-potsdam.de

Abstract. Automated service composition is an important approach to automatically aggregate existing functionality. While different planning algorithms are applied in this area, heuristic search is currently not used. Lacking features like the creation of compositions with parallel or alternative control flow are preventing its application. The prospect of using heuristic search for composition with quality of service properties motivated the extension of existing heuristic search algorithms. In this paper we present a heuristic search algorithm for automated service composition. Based on the requirements for automated service composition, shortcomings of existing algorithms are identified, and solutions for them presented.

Keywords: Processes and service composition, Process planning and flexible workflow

1 Introduction

Service Composition is an important approach to aggregate existing functionality into new functionality. Functionality, available as services, is composed and enacted as a process. Creating service compositions is a time-consuming, error-prone manual task. Hence, different approaches for its automation exist [1–4]. In this paper we present an approach for automated service composition based on a heuristic search algorithm.

Heuristic search is a promising approach for automated planning. Contrary to other planning approaches, it is currently not used for automated service composition. Lacking features prevent the wide-spread usage. But its clear-cut and easy to understand principle makes heuristic search a promising starting point for more elaborated automated service composition approaches. Metric-FF [5] allows planning with numerical properties and the optimization for them. This can be used to implement quality of service properties. Using heuristic search as the basis for semi-automated composition might also be a viable approach.

^{*} This paper presents results of the Adaptive Services Grid (ASG) project (contract number 004617, call identifier FP6-2003-IST-2) funded by the Sixth Framework Program of the European Commission.

But heuristic search algorithms have some severe shortcomings: Generated plans are totally-ordered. This is critical as it prevents taking advantage of possible parallelism and invoke services in parallel. Heuristic search also does not support alternative control flows. These are required if the exact services to invoke are only known at run-time, based on results of previously invoked services. For example, to process a payment if several different payment options are available the correct service must be invoked according to the payment option. If the payment option is selected only at run-time, the service composition must contain alternative control flows for each possible payment option. Constructing such plans is not supported by classical heuristic search planners.

In the following section a usage scenario is introduced. In section 3 the critical requirements that hinder the usage of heuristic search are presented. Section 4 then proposes extensions to overcome the shortcomings by extending an existing algorithm. The paper closes with a view on related work in section 5 and a conclusion.

2 Usage Scenario

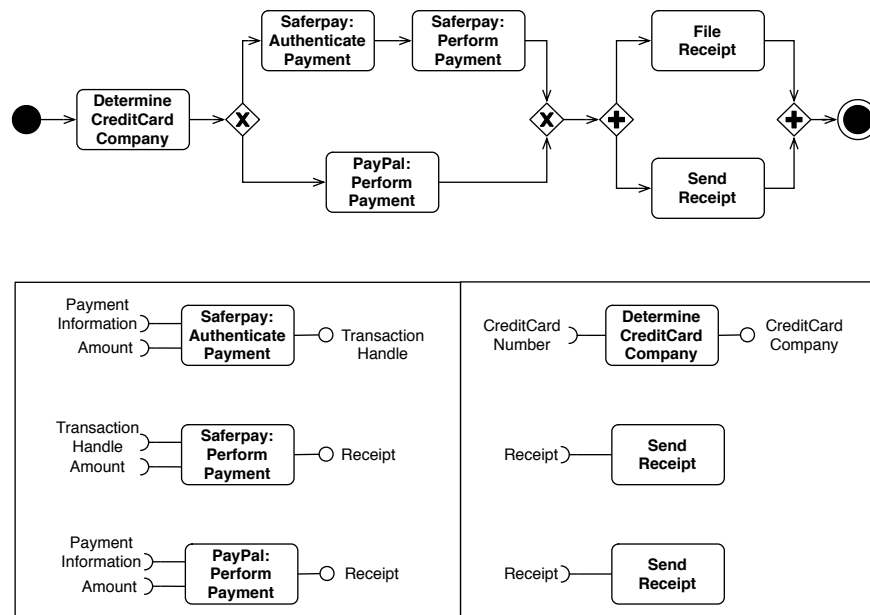


Fig. 1. Payment Composition

To illustrate the automated service composition approach of this paper a small usage scenario is introduced. It is a payment process that can be part of some larger business process. Figure 1 shows an example for a composition. It starts by determining the credit card company for a given credit card number. Afterwards the payment is performed with the correct payment service. The payment service of Saferpay¹ actually consists of two services. A payment has to be authenticated before it is performed. This is not necessary for the PayPal² payment service. Based on the issuing credit card company, payment through Saferpay, PayPal, or both is possible. Of course, always only one of them is actually used. Finally, the receipt is send to the customer and filed in the database. Based on the actual request, different compositions are possible. If for example the request already states that the credit card is from a specific credit card company supported by SaferPay, the composition only consists of four services: authenticate payment, perform payment, file receipt, and send receipt.

The presented composition algorithm is implemented and used as a part of the Adaptive Services Grid (ASG) project³. The scenario is derived from a larger scenario by one of the project partners. The Dynamic Supply Chain Management scenario is about the integration of suppliers in the domain of Internet Service Providers (ISP).

3 Shortcomings of Existing Heuristic Search Algorithms

In [6] we presented elaborated requirements analysis for automated service composition algorithms. Most of these requirements are fulfilled by existing heuristic search algorithms. Hence, we are limiting ourselves to the following critical ones:

1. Parallel control flow
2. Uncertainty in initial state and service effects
3. Alternative control flow
4. Creation of new variables

The first requirement is *parallel control flow*. Compositions consist of service invocations and their ordering. This ordering is the control flow. The straight forward approach is to assume a total ordering between service invocations and perform them sequentially. But in reality service invocations are often only partially ordered. If services do not depend on each other's results and do not conflict with each other, they can be invoked in parallel. This saves execution time. Therefore a composition algorithm must be able to create compositions with control flows that only contain the necessary orderings.

The second requirement is to support *uncertainty in the initial state and service effects*. Executing a service with uncertain effects leads to several new states. This is necessary to represent a service that determines the issuing credit

¹ Saferpay is a registered trademark of Telekurs Group.

² PayPal is a registered trademark of eBay Inc.

³ <http://asg-platform.org>

card company based on a credit card number. The exact outcome can only be determined after actually invoking the service for a given credit card number. After invoking a service with uncertain effects we are in more than one possible state. Hence, we might as well start with multiple possible states. Uncertainty in the initial state is necessary to express that for a certain fact only the possible values but not the exact value are known. For compositions containing service invocations with uncertain effects starting in an uncertain initial state it must be ensured that they work correctly in all possible situations.

The third requirement – *alternative control flow* – yields from the support of uncertainty in the initial state and in service effects. Invoking the service to determine the issuing credit card company based on the credit card number leads to several possible states. Based on the actual state, different service must be invoked to perform the payment. But determining the actual state can only be done when enacting the composition and invoking the services. To create compositions that work in all possible states it is necessary to support or-splits that lead to alternative control flows.

The fourth requirement – *creation of new variables* – results from the fact that in the data flow of a composition new data is created on the fly. This is not limited to writing the data into an existing variable but also includes the creation of new variables. This is complicated and often not possible in automated planning. This limitation of the planning model, already criticized in [7], simplifies planning. As all the variables are known, all possible service invocations can be calculated in advance. Services that are not invocable because the necessary variables for input or output parameters are missing, can be pruned.

Hence, in this planning model all variables used during composition must be defined in advance. This includes also intermediate variables that are neither used in the input nor in the output. For the payment scenario this means that a variable for the transaction handle of Saferpay must be defined. It is not obvious why such a variable could be necessary. By adding this variable we are encoding assumptions about possible composition results into the service request. For PayPal the transaction handle is not necessary and other companies might require other intermediate variables. Defining all necessary variables requires a lot of information about the service landscape and at least a rough idea of how the composition could look like (e.g. which services might be used). For a realistic composition approach it is therefore required that activities can create new variables and that the service composer takes these into account.

4 A Heuristic Search Algorithm for Service Composition

In this section a composition algorithm that overcomes all these limitations will be presented. Before starting with the description of the algorithm, the notions of service, service specification, service composition, and service request are introduced. A service is a discrete business functionality. It is described by a service specification:

Definition 1. A *service specification* $s = (\mathcal{I}, \mathcal{O}, p, e)$ is a tuple with

- \mathcal{I} : List of input parameters
- \mathcal{O} : List of output parameters
- p : The precondition of the service is a disjunction-free logical expression and must be satisfied in order to invoke the service.
- e : The effect of the service is a disjunction-free logical expression. It describes the changes to the current state resulting from the invocation of the service.

The afore-mentioned definition states that services have exactly one method to invoke. This differs from the service definition used for example in the WSDL standard [8]. But as WSDL does not specify choreographies, each method can be seen as an individual service and specified separately.

Definition 2. A *service request* $R = (a_0, g, D)$ is a triple consisting of the initial state a_0 , the goal state g and a service domain D . A state is a logical expression. This concept is refined later. A *service domain* $D = (\mathcal{S}, o)$ consists of a set of service specifications and an ontology describing the concepts used to specify services.

A *service composition* c is a list of service invocations $c = \langle s_1, \dots, s_k \rangle$. A service request is **fulfilled by** a service composition that starting from the initial state reaches a state that satisfies the goal state by subsequently invoking the services from the composition.

4.1 Enforced Hill-Climbing

Our algorithm is based on enforced hill-climbing [9]. It is a forward heuristic search in state space. A state is defined as follows:

Definition 3. A *logical expression* is defined as:

- A logical literal is a logical expression
- Two literals composed using the junctors \vee (disjunction) and \wedge (conjunction) is a logical expression

A logical expression is disjunction-free if it does not contain disjunctions. A disjunction-free logical expression a can be divided into the two logical expressions a^+ and a^- where a^+ contains all positive literals and a^- contains all negated literals. A **state** is a disjunction-free and negation-free logical expression.

A logical expression a **satisfies** another logical expression a' (written as: $a \models a'$) if every positive literal of a' is in a and no negative literal of a' is in a .

State space is the search space that is spanned by the states and the transitions in between them:

Definition 4. Service $s = (\mathcal{I}, \mathcal{O}, p, e)$ is **invokable** in state a if $a \models p$. **Invoking service** s in state a leads to a state transition. This can be defined by the state transition function $\gamma(a, s) = a'$. If $a \models p$ then $a' = a \cup e^+ \setminus \{x \mid \neg x \in e^-\}$

A state a has a direct successor a' , written as $a \rightarrow a'$, if a service s exists and $\gamma(a, s) = a'$. The successor relation can be inductively extended to indirect successors $\rightarrow^+ = \rightarrow \cup \{(a, a'') \mid (a, a') \in \rightarrow \wedge (a', a'') \in \rightarrow^+\}$

Enforced Hill-Climbing is an extension of Hill-Climbing. Hill-Climbing uses a heuristic function $h(a, g)$ to select states until the goal is reached. The heuristic $h(a, g)$ delivers an approximation of the distance of the state a to the goal g . Starting with the initial state, a new state is selected from the direct successors. The first successor that is, according to the heuristic, better than the current state is selected and assigned as the new current state. This process is continued until the current state satisfies the goal state. Given an admissible heuristic and a mechanism to prevent visiting states multiple times, the algorithm always terminates. It terminates successfully if it reaches a state that satisfies the goal state. It fails if a state a is reached so that no direct successor a' with $h(a', g) < h(a, g)$ exists.

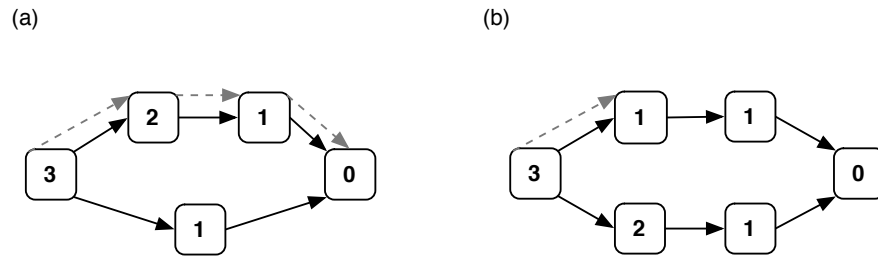


Fig. 2. Hill Climbing is not optimal (a) and incomplete (b)

Hill-Climbing does not create optimal compositions and it is incomplete. Figure 2(a) illustrates the reason for the in-optimality. Displayed are states, their heuristic values, and possible state transitions. If the state with heuristic 2 is evaluated first, it is selected even though a shorter path exists. Another problem is the greediness of Hill-Climbing. Greediness means that optimization is done locally without taking the path to the current state into account. This is only of importance if a cost function is associated with state transitions. Otherwise the admissible heuristic guarantees that greediness does not affect the composition result. Figure 2(b) demonstrates why Hill-Climbing is incomplete: If the upper path is taken, composition fails after the first state with heuristic 1 as no direct successor with a better heuristic can be found. Such a state is called a local maximum.

Enforced Hill-Climbing solves the problem of local maxima by switching to breadth-first search if it gets trapped in a local maximum. This works as depicted in Fig. 3. If the evaluation of a state shows that it is not better than the current states all its direct successor are added to the end of A' . Hence when all direct successors are evaluated and none was better than the current state, Enforced Hill-Climbing starts evaluating the successors of the successors. This is continued until either a better state is found or no reachable states are unevaluated and composition fails. In the situation from Fig. 2(c) Enforced Hill-Climbing switches to breadth-first search in the state with no better direct successors.

```

1   $a = i$ 
2   $c := \text{empty composition}$ 
3  while  $\neg(a \models g)$ 
4       $A' = \text{new Queue}$ 
5      enqueue( $A', \{a' | a \rightarrow a'\}$ )
6      for  $a' \in A'$ 
7          if  $h(a', g) < h(a, g)$ 
8              add( $c, s$ ) with  $\gamma(a, s) = a'$ 
9               $s = s'$ 
10             goto 3
11         else
12             enqueue( $A', \{a'' | a' \rightarrow a''\}$ )
13         end
14     end
15     composition failed
16 end
17 composition successful

```

Fig. 3. Enforced Hill-Climbing

Through breadth-first search the state with heuristic 0 (the goal) is found and it can finish successfully. Regardless of this extension is Enforced Hill-Climbing still incomplete but termination is still guaranteed as breadth-first search always terminates. Fig. 4 shows that composition fails if the upper path is taken. The upper path is a dead end and the algorithm is not able to turn around and leave it. As termination is always guaranteed, one approach to deal with incompleteness, as proposed by [9], is to switch to another complete but slower search algorithm (e.g. A^*) if Enforced Hill-Climbing fails. The enforcement extension of Hill-Climbing does not affect the in-optimality of the algorithm.

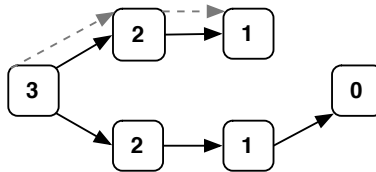


Fig. 4. Enforced Hill-Climbing is incomplete

4.2 Extending Enforced Hill-Climbing

Enforced Hill-Climbing does not support any of the aforementioned requirements. Uncertain effects or initial states cannot be handled by creating alternative control flows. Compositions are strictly sequential and no variables can be

created during the composition. In the following we present how each requirement can be addressed.

Implementing requirement 1: Parallel control flow The first step towards parallel control flow is to support the parallel selection of multiple services. Figure 5 illustrates that this leads to a denser search space as more state transitions are possible. But at the same time paths become shorter.

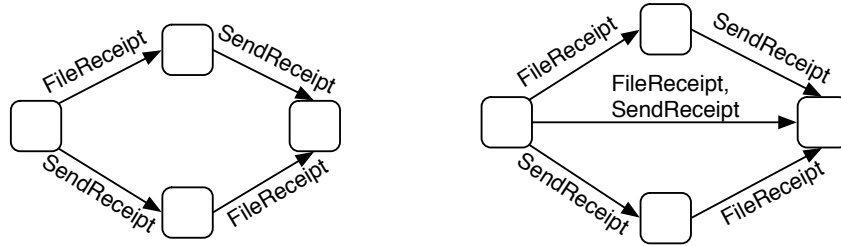


Fig. 5. State Space without and with parallel selection

To invoke services in parallel it must be ensured that they can actually work in parallel. First this means that services where one service creates the precondition of the other service cannot be invoked in parallel. For example, the payment processing must be finished before the receipt can be send. This can be ensured by extending the invocability definition to sets of services: a set of services is invocable in a given state if every service is invocable in the state. But this definition is not sufficient as two invocable services may be in conflict. Before we can define invocability for sets of services we need to define what it means if two services are in conflict:

Definition 5. Two services $s_1 = (\mathcal{I}_1, \mathcal{O}_1, p_1, e_1)$ and $s_2 = (\mathcal{I}_2, \mathcal{O}_2, p_2, e_2)$ are in **conflict** if:

- s_1 deletes the precondition of s_2 : $\neg x \in e_1 \wedge x \in p_2$
- s_1 creates a fact whose negation is the precondition of s_2 : $x \in e_1 \wedge \neg x \in p_2$
- s_1 and s_2 have inconsistent effects: $x \in e_1 \wedge \neg x \in e_2$

A set of services $\mathcal{S} = \{s_1, \dots, s_n\}$ is in conflict if two services $s_i (1 \leq i \leq n)$ and $s_j (1 \leq j \leq n)$ exists which are in conflict.

Based on this notion we can define invocability and invocation for service sets:

Definition 6. A set of services is **invokable** if each service is invocable and it is conflict-free. Given a set of conflict-free services $\mathcal{S} = \{s_1, \dots, s_n\}$ **invocation** of \mathcal{S} is equal to the sequential invocation of all $s_i (1 \leq i \leq n)$ in arbitrary order. The state transition function can be extended accordingly: $\gamma(a, \mathcal{S}) = a'$

To support the parallel selection of multiple services one modification of Enforced Hill-Climbing is necessary: Line 8 where the new service is added to the composition must deal with the extended state transition function $\gamma(a, \mathcal{S})$. More than one service can be added to a composition at the same time. As the parallel selection should be reflected in the resulting composition, we need to modify our composition definition. The easiest way to do that would be to extend the previous list of services to a list of service sets. But with respect to further additions we choose another definition:

Definition 7. A composition $C = (\mathcal{S}, \prec^{cond})$ consists of a set of service invocations \mathcal{S} and a partial order \prec^{cond} between them. For two services $s_i, s_j \in \mathcal{S}$ an ordering $s_i \prec^{cond} s_j$ is defined if s_i was added to the composition before s_j . Here $cond$ is that part of the effect of s_i that is necessary to invoke s_j . Likewise, $s_i \not\prec^{cond} s_j$ if both were added in the same step.

Implementing requirement 2: Uncertainty in initial state and service effects States, preconditions, and effects must include disjunction to support uncertainty. Disjunction in states is not only used to express uncertainty about the initial state. It also used to express several distinct goal states. Disjunction in the precondition of a service allows to express that the service is invocable in different situations. This does not increase the expressiveness as this can be simulated by multiple services. Disjunction in service effects can be used to express uncertainty about the service's outcome. To work with these richer expressions, we introduce a set-based representation of logical expressions with disjunctions:

Definition 8. Given a logical expression a its disjunctive normal form can be expressed as a set $a_{set} = \{a_1, \dots, a_n\}$ of disjunction-free logical expressions. Here each a_i represents one conjunction of the disjunctive normal form.

A logical expression and its set-based representation can be used interchangeably. When a distinction is necessary we will name the set-based notation a_{set} . When speaking about a state and its set-based representation it is helpful to think of the set-based representation as a set of possible states. The definition for state satisfaction needs to be extended accordingly:

Definition 9. A state a **satisfies** another state g if $\forall a_i \in a_{set} \exists g_j \in g_{set} a_i \models g_j$.

Hence, a set of possible current states satisfies a set of allowed goal state if every possible current state satisfies at least one allowed goal states. Now we have developed the foundation to represent uncertainty. Yet it is unclear how we can actually deal with uncertainty during planning. In automated planning two approaches have been developed: conformant planning and contingent planning. Using conformant planning, additional service invocations are added that ensure

the correct working of the composition, without actually determining the current state or the actual effects of service invocations. While this is a simple model, it is often not practicable. For example instead of first determining the correct credit card company and then charging the credit card only with the correct payment service, it is tried to charge the credit card using each payment service. While, hopefully, the credit card is only charged once, the other services may charge a fee making the payment process very expensive. Conformant planning makes most sense when controlling robots that lack sensors. In business scenarios another approach is more practicable. Contingent planning introduces the ability to sense the actual value of fact during run-time and then continue accordingly. This means after determining the credit card company for a credit card, the actual value is sensed during run-time and then the correct service is invoked. For the control flow of the composition or-splits must be support that lead to alternative control flows.

Implementing requirement 3: Alternative control flow In the previous section we extended the notion of states to include uncertainty. Service effects can now include disjunction as well. This means that we can actually reach several alternative states by invoking a service. To support contingent planning it must also be possible to invoke a service if it is only invocable in some of the current states. Figure 6 illustrates this situation. Invoking the service to determine the credit card company leads here to two possible states⁴. In the first state the SaferPay authentication service is invocable and in the second state the PayPal payment service is invocable. Invoking them only changes the state in which they were invocable. As multiple services may be selected (see section 4.2) both services can be selected in parallel changing both states at once. To support this notion, invocation and invocability need to be extended:

Definition 10. A service $s = (\mathcal{I}, \mathcal{O}, p, e)$ is *invokable* in a state a if $\exists a_i \in a_{set} \exists p_j \in p_{set} a_i \models p_j$. *Invoking a service* in a state a leads to a state transition. This can be defined by a state transition function $\gamma(a, s) = a'$. If $a \models p$ then $a' = \{a_i | a_i \in a_{set}, \forall p_j \in p_{set}, a_i \not\models p_j\} \cup \{a_i \circ e | a_i \in a_{set}, \exists p_j \in p_{set}, a_i \models p_j\}$. The operation $a_i \circ e = \{a_i \cup e_j^+ \setminus \{x | \neg x \in e_j^-\} | e_j \in e_{set}\}$ applies the effect to one logical expression.

Invoking a service with uncertain effects results in several possible states. If subsequent services cannot be invoked in all states, an or-split is added to the composition. In our example this is the case after determining the credit card company.

For our composition algorithm it is irrelevant which path from Fig. 6 is actually taken, because only necessary orderings between service invocations are added. This is done by linking two service invocations only if one produces the precondition of the other or if they are in conflict. Formally:

⁴ In reality this might be more, but two is sufficient for presentation.

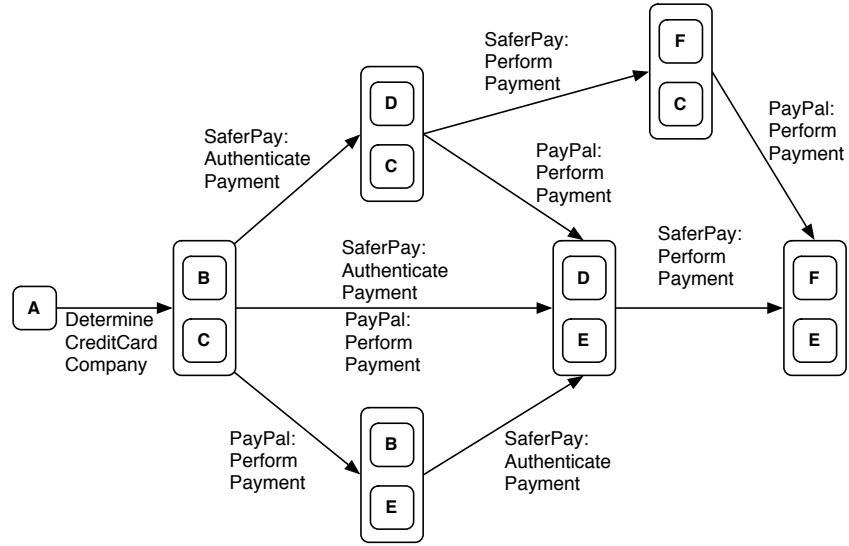


Fig. 6. Extended State Transition

Definition 11. For two services $s_1 = (\mathcal{I}_1, \mathcal{O}_1, p_1, e_1)$ and $s_2 = (\mathcal{I}_2, \mathcal{O}_2, p_2, e_2)$ a link $s_1 \stackrel{cond}{\prec} s_2$ exists if:

- $cond \in p_2 \wedge producer(s_1, a, x) \wedge x \in cond$ where $producer(s, a, x)$ is the relation of fact x from state a produced by service s
- or s_1 and s_2 are in conflict.

Often it is not only necessary to create alternative branches but to also merge them later. In our example this is necessary after payment has been performed. A first approach to merging might be to detect equivalent states and unify them to one state. In Fig. 6 states E and F seem to be mergable because they represent the same fact: payment has been performed. In reality things are not that easy and calculating state equivalence is hard and may be impossible. As a matter of fact E and F are not really equivalent as F also includes the transaction handle. Although both states mean the same for us, detecting this is not possible. We can only merge states which are exactly identical. This is unproblematic as in the end we are not interested in merging states but merging control flows. This is a lot easier: Control flows can be merged if the current set of services is invocable in some or all possible states. Both succeeding services in our example – filing the receipt and sending the receipt to the customer – can be invoked in both possible states. Hence we can merge the alternative control flows. The inability to merge the states costs us performance, as we have to evaluate more states, but it does not prevent us from doing a merge.

The interesting point about introducing only necessary links is that it renders the parallel selection of service unnecessary. As only necessary links are added,

two service that can be invoked in parallel will be composed as running in parallel even if they are selected subsequently. We are still using the parallel selection as it is currently unclear whether its denser search space is a disadvantage or its shorter search paths are an advantage.

Implementing requirement 4: Creation of new variables Creating new variables is currently not supported by most planners. This results not only from limitations of the language used to describe requests [7] but it also greatly simplifies creating the composition. If all variables are known in advance it is easy to determine which services can be invoked. If we do not specify that a transaction handle variable exists, the SaferPay services are never invocable and hence can be pruned. But as we do not want to specify the transaction handle in our request, this behavior is undesirable.

To solve this problem we need to allow the creation of new variables if a matching variable for the output of a service does not exist. But the unrestricted addition is problematic as this yields a possibly infinite set of states and makes planning semi-decidable [10, 11]. Thus we are introducing a very restricted form of variable creation. A variable may only be created if no variable of the same type already exists. While this keeps the problem decidable it may be too restrictive as it fails if two variables of the same type need to be created. We are currently not allowing the deletion of variables, as we have not encountered any practical use for it.

4.3 A Heuristic for Extended Enforced Hill-Climbing

As the heuristic guides the search it is crucial for the performance of the composer. An approach to find a heuristic for a given problem is to relax it (make it simpler). Enforced Hill-Climbing was originally developed together with the *Relaxed Graphplan* heuristic [9]. Essentially it solves a simplified version of the composition request using the Graphplan planning algorithm [12]. We take the length of the generated composition as the heuristic. Graphplan works by first creating a planning graph and then extracting the solution from it. The relaxation or simplification of the problem results from ignoring the negative effects of service invocations. In the presence of negative effects back tracking is necessary during solution extraction. As negative effects are ignored, the heuristic can be calculated in polynomial time [9].

For our algorithm the calculation of the heuristic function can be even more simplified. We skip the solution extraction phase. Why the solution extraction phase was necessary for FF and can be skipped here, will become evident after we have build up such a planning graph. A planning graph consists of two kinds of nodes: fact nodes and activity nodes. Fact nodes represent literals (or facts) from states and activities represent service invocations. Starting from the facts of the initial state alternating layers of facts and activities are added until a fact layer is reached that satisfies the goal. Figure 7 illustrates such a graph. Starting from the initial state all invocable services (in this case *s1* and *s2*) are

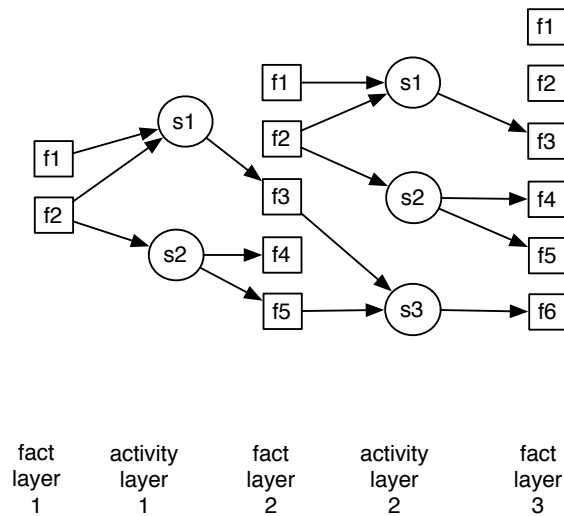


Fig. 7. A Planning Graph

added to the first activity layer. This activity layer *produces* a new fact layer including all the positive effects of $s1$ and $s2$. Now a new service $s3$ is invocable. The resulting fact layer now includes our goal (e.g. $f6$) and we are finished with building the graph. The original planning graph from Graphplan additionally contains mutual exclusion relations between two activities or two facts if the activities are in conflict or if the facts only result from conflicting activities. If negative effects are ignored no mutual exclusion relations will be added as all conflicts emerge from negative effects. Now the original Relaxed Graphplan heuristic would continue by extracting a sequential solution from the graph. But as we are not interested in the length of a sequential solution we can directly count the number of activity layers and take it as our heuristic. As the resulting composition contains as much parallel invocation of services as possible this is quite a good. The upper-limit for the heuristic is the actual distance from the goal. The value can actually be lower as we are ignoring conflicts. Hence it is admissible.

Like the original Relaxed Graphplan heuristic, this heuristic can be calculated in polynomial time. To reduce space consumption we can use an optimization. As we are only interested in the number of activity layers and we do not want to extract a solution, it is sufficient to just keep the current fact layer and count the number of activity layers used to reach this fact layer. This greatly reduces space consumption.

5 Related Work

As mentioned in the introduction, a lot of different approaches towards service composition exist. Most are adapting existing automated planning algorithms. In [1] a slightly different approach is followed. They designed their own algorithm that finds the necessary services to invoke through backward-chaining and then identifies additional necessary services in a second forward-chaining phase. In [3] Hierarchical Task Network (HTN) planning is used for service composition. HTN planning is based on the notion of composite tasks that can be refined to atomic tasks using predefined methods. In domains where these methods that are essentially sub-processes exist, HTN planning provides a very fast approach. In [2] the model checking planner MBP is presented. Model checking is based on nondeterministic state-transition systems. States are not represented explicitly. It has the ability to generate conformant and contingent plans. It can also generate cyclic plans and has the notion of extended goals. Through extended goals it is possible to impose requirements not only on the goal state but also on intermediate states. But it is not able to create compositions with parallel control flows as the definition of state transition systems is restricted to invoking just one service per state transition.

We are using heuristic search instead of any of the above-mentioned approaches as heuristic search promises to be easily extensible to support optimization for QoS properties and the adaption towards semi-automated composition. Heuristic search algorithms are currently not used for automated service composition. Our work is based upon previous research by Hoffmann and Nebel who developed the planners FF [9] and Metric-FF [5]. They introduced Enforced Hill-Climbing and Relaxed Graphplan as a heuristic. Metric-FF also supports numerical properties and the optimization for them. This functionality can be used to optimize for QoS properties. As demonstrated earlier their algorithm does not support uncertainty about the initial state or service invocation effects, is not able to compose parallel or alternative control flows, and does not create intermediate variables.

Recently, several extensions to heuristic search algorithms were proposed to support some of the required features [13–16]. But all of them are based on the restricted planning model imposed by the Planning Domain Description Language (PDDL) and thus are not able to create intermediate variables [7]. LPG [13] performs heuristic search in plan space instead of state space. The nodes of the search space are (partial) plans and transitions between them are plan refinement operations (e.g.: adding an additional service invocation). LPG is a temporal planner and hence supports parallel control flow. Compositions are partially ordered and durations are assigned to service invocations. LPG supports optimization for duration and other numerical properties. It can not deal with uncertainty and it cannot create alternative control flows. Sapa [14] is also a temporal planner and supports optimization for duration and numerical properties. But unlike LPG it does perform search in state space. In that regard it is very similar to FF and Metric-FF. Sapa uses A* as the search strategy. In contrast to Enforced Hill-Climbing is A* complete and optimal if an admissible

heuristic is used. We did not use A* because you have to trade in performance for completeness and optimality. Sapa does not support uncertainty and the creation of alternative control flows. Conformant-FF [15] and Contingent-FF [16] are both extension of the original FF planner. They extend it by functionality for conformant planning and contingent planning. Both work with uncertainty through the notion of *belief states*. A belief state is equivalent to our extended state definition and incorporates a set disjunction-free states. It represents the possible states. For Conformant-FF the main difference to FF is the handling of the belief states: Planning starts in a set of possible states and is finished if all the possible current states satisfy the goal. It creates conformant plans without alternative control flows and is therefore not usable for automated service composition. Contingent-FF on the other hand creates contingent plans that include alternative control flows. It is quite similar to our approach. Through its more efficient representation of possible states and further optimizations it has some advantages over our approach. But it does currently not support parallel control flow and alternative control flows are not merged resulting in tree-shaped compositions.

6 Conclusion

In this paper we presented a heuristic search algorithm for automated service composition. It supports the creation of composition with parallel and alternative control flows allowing uncertainty about the initial state and service effects. The ability to create variables during composition ensures its applicability in real world business scenarios. Our implementation currently significantly is slower than FF and its descendants. There are two main reasons for this: a larger search space and missing optimizations. The search space is larger than the one of FF as we search for possible parallel invocations and create intermediate variables on the fly. Several different optimization strategies for heuristic search algorithms have been proposed. With *helpful actions* [9] or *favored actions*[17] a subset of the invocable services representing the most promising ones is defined. Evaluating them first often considerably increases performance. In Conformant-FF and Contingent-FF belief states are represented by the initial state and the invoked service sequence reducing efforts to calculating the actual state. Adapting these optimizations to our approach will increase performance significantly.

Future directions of our research will be directed at extending the composer by numerical state properties. This extension allows for the representation of Quality of Service properties (e.g. price, execution time). Using these properties during composition makes it possible to optimize for desired values. We are also adapting the search algorithm to work in a semi-automated modeling environment. Here a human modeler creates the composition, but the composition component assists him by finding matching services / sub-compositions or verifying that a composition fulfills a given goal.

References

1. Zeng, L., Benatallah, B., Lei, H., Ngu, A., Flaxer, D., Chang, H.: Flexible Composition of Enterprise Web Services. *Electronic Markets – Web Services* **13** (2003) 141–152
2. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and monitoring web service composition. In: *Workshop on Planning and Scheduling for Web and Grid Services (held in conjunction with The 14th International Conference on Automated Planning and Scheduling)*. (2004) 70 – 71
3. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using shop2. *Journal of Web Semantics* **1** (2004) 377–396
4. Berardi, D., Calvanese, D., Giacomo, G.D., Mecella, M.: Composition of services with nondeterministic observable behaviour. In: *Proceedings of the Third International Conference on Service-Oriented Computing*. Volume 3826 of *Lecture Notes In Computer Science.*, Heidelberg (2005) 520–526
5. Hoffmann, J.: Metric-FF planning system: Translating "ignoring delete lists" to numeric state variables. *Journal Of Artificial Intelligence Research* **20** (2003) 291 – 341
6. Meyer, H., Kuropka, D.: Requirements for automated service composition. In Eder, J., Dustdar, S., eds.: *Business Process Management Workshops*. Volume 4103 of *Lecture Notes In Computer Science.*, Heidelberg, Springer (2006) (to appear).
7. Boddy, M.: Imperfect match: PDDL 2.1 and real applications. *Journal Of Artificial Intelligence Research* **20** (2003) 133 – 137
8. W3C: *Web Services Description Language (WSDL) 1.1*. (2001)
9. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* **14** (2001) 253 – 302
10. Chapman, D.: Planning for conjunctive goals. *Artificial Intelligence* **32** (1987) 333–377
11. Erol, K., Nau, D.S., Subrahmanian, V.: Complexity, decidability and undecidability results for domain-independent planning. *Artificial Intelligence* **76** (1995) 75–88
12. Blum, A., Furst, M.: Fast planning through planning graph analysis. *Artificial Intelligence* **90** (1997) 281–300
13. Gerevini, A., Saetti, A., Serina, I.: Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research* **20** (2003) 239 – 290
14. Do, M., Kambhampati, S.: Sapa: A multi-objective metric temporal planner. *Journal Of Artificial Intelligence Research* **20** (2003) 155 – 194
15. Brafman, R., Hoffmann, J.: Conformant planning via heuristic forward search: A new approach. In Sven Koenig, Shlomo Zilbe Koenig, S.Z., ed.: *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, Morgan-Kaufmann (2004) 355 – 364
16. Hoffmann, J., Brafman, R.: Contingent planning via heuristic forward search with implicit belief states. In: *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, Morgan-Kaufmann (2005)
17. McDermott, D.: A heuristic estimator for means-ends analysis in planning. In: *Proceedings of the International Conference on Artificial Intelligence Planning Systems*. (1996) 142–149