A Semi-automated Orchestration Tool for Service-based Business Processes

Jan Schaffner[1], Harald Meyer[2], and Cafer Tosun[1]

[1] SAP Labs, Inc.
3421 Hillview Ave, Palo Alto, CA 94304, USA
{jan.schaffner,cafer.tosun}@sap.com
[2] Hasso-Plattner-Institute for IT-Systems-Engineering at the University of Potsdam
Prof.-Dr.-Helmert-Strasse 2-3, 14482 Potsdam, Germany
harald.meyer@hpi.uni-potsdam.de

Abstract. When creating service compositions from a very large number of atomic service operations, it is inherently difficult for the modeler to discover suitable operations for his particular goal. Automated service composition claims to solve this problem, but only works if complete and correct ontologies alongside with service descriptions are in place.
In this paper, we present a semi-automated modeling environment for Web service compositions. At every step in the process of creating the composition, the environment suggests the modeler a number of relevant Web services. Furthermore, the environment summarizes the problems that would prevent the composed service from being invocable. The environment is also able to insert composed services into the composition at suitable places, with atomic services producing the required data artifacts to come to an invocable composition.
Our results show that this mixed initiative approach significantly eases the creation of composed services. We validated our implementation with the leading vendor of business applications, using their processes and service repository, which spans across multiple functional areas of enterprise computing.

1 Introduction

In recent years, the fact that handcrafted service compositions are often erroneous has been serving as a rationale to automate the creation of Web service compositions ([1–4]). Academia has proposed systems that automatically create invocable plans for each individual case at runtime. This opposes the idea of creating composed services that cover as many cases as possible. The plans are produced in a fully automatic fashion, based on domain knowledge and semantic service descriptions. While automated planners are able to reduce complexity, inflexibility and error-proneness akin to the creation of composed services, several drawbacks can be identified: Automated planning relies on the availability of complete formal representations of the domain knowledge. The task of formally specifying a domain in sufficient fidelity presents a tremendous challenge. Especially for complex domains we can legitimately assume that complete ontologies

will not be available in the near future. Incomplete domain knowledge, however, will often result in the situation that an automated planner fails to produce a plan. Erroneous domain knowledge, moreover, can result in situations where a planner finds wrong plans. In contrast, a human planner can draw upon his experience within a specific domain. This experience will often compensate for missing or erroneous ontologies. Moreover, the fact that fully automated service composition methods do not require a human in the loop poses an organizational and juridical impediment: In business reality, it is required that concrete persons are responsible for a particular business process. This fact lowers the industry acceptance of automated planning techniques; their transition from research to industry is progressing slowly.

In this paper we show that the techniques from automated planning can be used to ease the manual creation of business processes. The incorporation of matchmaking technologies used by automated planners into a semi-automated modeling tool for creating enterprise service compositions has several advantages. The problems manual service composition can be reduced or eliminated by the aid of new "mixed initiative features". This paper is organized as follows: Section 2 presents a scenario from practice, which is used in section 3 to show the usefulness of the proposed mixed initiative features. Section 3 also discusses the implementation of these mixed initiative features. In section 4 we give an overview of related work in the field of semi-automated service composition and discuss our tool from that perspective. Section 5 concludes the paper.

2 Usage Scenario: Leave Request

The following scenario is taken from Duet[1], a recent software product by SAP and Microsoft. Duet extends the reach of SAP's business processes to a large number of users by embedding them into Microsoft's Office environment. We extracted the process flow among the ERP Web services Duet is built upon. We are using the process as a case study for the semi-automated composition environment developed in Jan Schaffner's Master's thesis [**?**]. The leave request scenario consists of two parts: First, an employee files a leave request. Second, his manager approves or denies this request. Therefore, the two roles "employee" and "manager" participate in the leave request process. Due to length constraints, we limit ourselves to the part of the process in the role of the employee. The scenario is depicted in figure 1 using the Business Process Modeling Notation (BPMN [5]). The activities in the diagram correspond to Web service operations. To describe the semantics of the operations, we extended the BPMN syntax so that we can depict WSMO service capabilities: The inputs that a service consumes and the conditions over these inputs make up the "precondition". The outputs of a service and the relation between input and output make up the "postcondition".

Before the employee files a leave request, he will typically try to get an overview of his time balance and suitable dates for the leave. Duet will collect

[1] http://www.duet.com

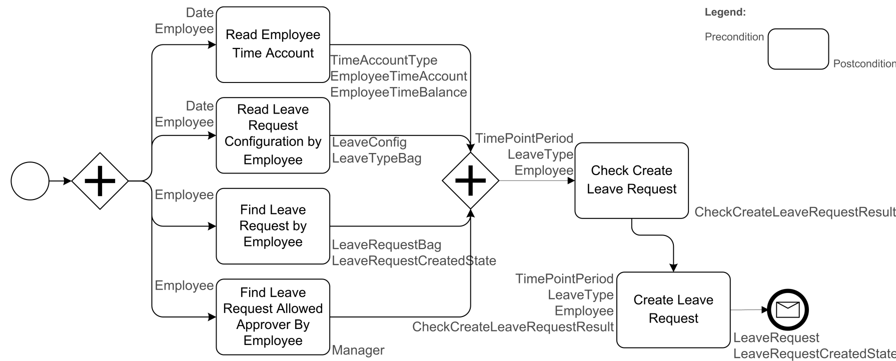

Fig. 1. Leave request scenario

this information when the leave request application pane is opened. Therefore, Duet will call the following four Web service operations.

– **Read Employee Time Account** This operation returns the time balance of an employee consisting of paid vacation, overtime, and sick leaves. The operation takes an employee object uniquely representing the employee and a key date, for which the balances are returned, as input.
– **Read Leave Request Configuration by Employee** This operation outputs the leave configuration (allowed leaves such as half or full day) for a specific employee as stored in the ERP system. The operation takes an employee object and a key date as input.
– **Find Leave Request By Employee** It might be the case that an employee has recently filed other leave requests which are not yet processed. This operation returns an employee's pending leave requests, so that he or she can consider them together with the time balance. The operation takes an employee object as input.
– **Find Leave Request Allowed Approver by Employee** A leave request is approved by the line manager of the employee filing it. In some cases, a different approver or no approval at all is necessary. This operation returns the employee object corresponding to the allowed approver for the leave request. It takes an employee object as input.

The information retrieved by the four service operation described above is visualized in Duet and the employee can decide on a day or a period browsing his Outlook calendar. This yields the sequential invocation of the two following operations:

– **Check Create Leave Request** Before a leave request is created in the ERP system, it must be checked for plausibility. This operation takes the same inputs as the operation that creates the leave request, which are an employee object, the leave period and the leave type. If the check is successful, the operation returns a positive result.

– **Create Leave Request** After the plausibility check has been successful, this operation finally creates the leave request in the ERP system. As a result, a leave request is created.

3 Mixed Initiative Features

In this section, we will describe three mixed initiative features which are characteristic for semi-automated service composition. The leave request business process from the previous section will be constructed step by step supporting and motivating the three features. We will then present the realization of each feature in detail. A more general introduction of the mixed initiative features for semi-automated composition can be found in [6].

3.1 Filter Inappropriate Services

The number of service operations that are available as building blocks for the composition can be extremely high. In the context of SAP, for example, the central repository contains more than 1000 services. This results in a complexity that is hard to oversee. Especially if compositions are to be realized by users from a non-technical background, a modeling tool for service compositions should filter the set of available services. Such filtering can be done based on semantic service descriptions.

Business Scenario When the leave request is to be created from scratch, the tool will first retrieve all available Web services. The modeler starts out with adding the role “employee” to the composition by selecting this role from a list of all available roles (e.g., “supplier”, “customer”, “manager”). Our tool then assumes the implicit availability of a variable of the complex type “employee”, representing the person who takes part in the business process in this role. The tool is now able to filter the list of available service operations to those that require an employee object as an input. Our experiments have shown that filtering all service operations that are not be invocable in the current step of the composition is too strict. The tool therefore also presents service operations that are nearly invocable in the sense that only one input is missing. Using this technique, we are able to retrieve very reasonable suggestions from SAP’s service repository. The operations in this repository are grouped around so-called enterprise services. In our example, the modeler would therefore now expand the “Time and Leave Management” enterprise service and select the first four operations depicted in figure 1. As there are no dependencies between the activities, the user connects the operations using a parallel control flow. This is shown in figure 2.

At this point, the modeler is able to retrieve more service suggestions through the filtering mechanism by clicking on the merge node of the parallel split. Our tool will then present a list of service operations that are invocable or nearly invocable based on the union all postconditions of the services which are in

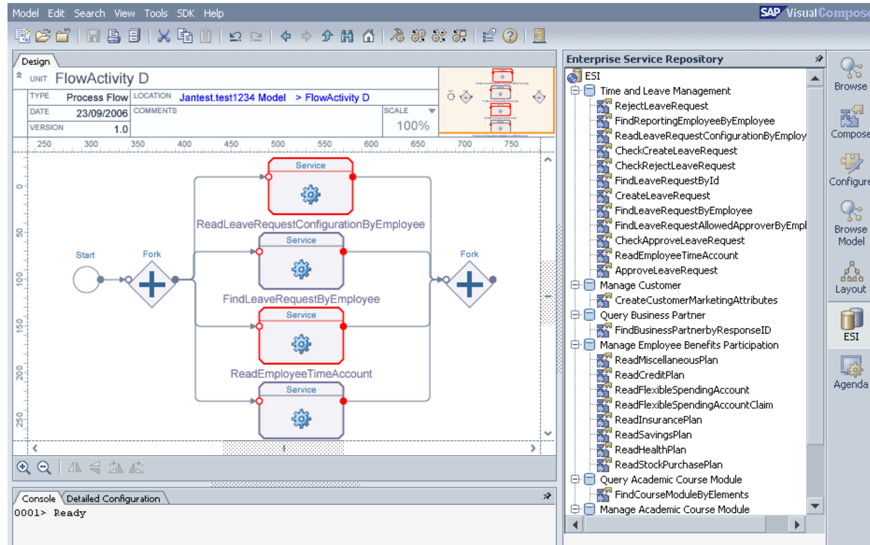

Fig. 2. Screenshot of the modeling tool

the composition so far. The postconditions (i.e., the output data types) of the operations are also depicted in figure 1. Amongst others, our tool will suggest the operation *Check Create Leave Request* as a nearly invocable service. The modeler adds it to the composition and creates a link between the merge node and the operation.

Realization At each step of the composition (i.e., each change in the process by the modeler) the state of the composition is translated into a query against our semi-automated service composition engine, which is built on top of a WSML reasoner. In this context, the term "state" refers to the postconditions of all service operations that are currently in the composition. The currently selected role (e.g., "employee") is also added to the state. As already mentioned in the business scenario, our tool suggests both invocable services and nearly invocable services. Therefore, two corresponding methods are provided by our semi-automated composition engine.

The first method returns a list of invocable service operations, alongside their pre- and postconditions, possible roles in which they can be executed, the default role, possible variable bindings for the current state of the composition, a default variable binding and the enterprise service they belong to. Moreover, this list is ordered by relevance for the user in the current context. Listing 1.1 presents our algorithm for finding and weighting invocable services for a given state in pseudo code.

```
1 findInvocableServicesOrdered(State state, String role)
    register state with reasoner;
```

```
3    retrieve list of registered operations from reasoner;
     for each registered operation do
5      if operation is invocable do
         compute match distance for each binding;
7        store binding with lowest match distance as default binding;
         store all other bindings, preconditions and postconditions;
9    unregister state with reasoner;
     compute weightings based on lowest match distances;
11   for each invocable operation do
       if NFP specifying an intended roles for the operation exists
13       if NFP matches role
           increase weighting by 1;
15   order operations by weighting;
     return ordered list of operations;
```

Listing 1.1. Compute ordered list of invocable service operations

The match distance in line 6 of listing 1.1 is computed based on the distance of the data types in the variable bindings and the data types specified in the preconditions of the operations. As an example, we consider an operation that consumes a parameter "employee" as a precondition. Additionally to the "employee" concept there as subconcept of "employee" called "manager". Given a state with an "employee" instance the match distance between a variable binding and the service's precondition is 0. But if we only have a "manager" instance, the distance between the variable binding and the precondition is 1. Because the first binding has a lower match distance than the second, it is the default binding of the operation in the current state. If a precondition requires more than one data type, the match distance is aggregated over all variables in a binding.

The weightings for the operations (line 10 listing 1.1) are based on the match distances: For each operation, we take variable binding with the lowest match distance (i.e., the default binding) into account. The operation corresponding to the binding with the highest match distance gets the lowest weighting, and so forth. To further differentiate the relevance of the discovered services, the role of the current state of the composition is compared to the intended role of each invocable operation. An intended role is the role for which an operation has been designed. For example, the operation *Create Leave Request* is designed for the role "employee", while it is also invocable for composition states with the role "manager". If the role of the current composition state and the intended role of an operation match, the weighting of the operation is increased.

The second method provided by our semi-automated service composition engines discovers service operations that are nearly invocable in the current state of the composition. An operation is nearly invocable, if only one input parameter is missing. Listing 1.2 lists the pseudo code of this method. We traverse the ontology, add each concept to the current composition state one after another, and check for invocable operations in the modified state. Please note that there is no ranking in case the method returns multiple nearly invocable operations. This allows for an optimization in the process of discovering the nearly invocable services: We use the most specific subconcept for each concept in the ontology

before we search for invocable services in the modified state. In doing so, we only have to perform this search operation for a subset of the concepts in the ontology, which improves the response time.

```
   findNearlyInvocableServices(State state)
 2   operations = findInvocableServices(state);
     for each concept in the ontology do
 4     if concept is not marked as visited
         sc = findMostSpecificConcepts(c);
 6       for each concept s in sc
           add s to state;
 8         register state with reasoner;
           nOps = findInvocableServices(state);
10         add nOps − operations to result;
           deregister state with reasoner;
12         remove s from state;
           mark s as visited;
14   return list of nearly invocable operations (result);

16 Concept[] findMostSpecificConcepts(Concept c)
     if c has subconcepts
18     sc = new Concept[];
       for each subconcept s of c
20       sc += findMostSpecificConcept(s);
       return sc;
22   else return [c];
```

Listing 1.2. Compute list of nearly invocable service operations

3.2 Check Validity

As the human modeler has full control over the modeling, it is possible that he introduces errors. It is therefore necessary to be able to check the semantic validity of the process. As opposed to syntactic validity checking structural correctness criteria like soundness [7], semantic validity bases on semantic descriptions for individual activities to define correctness of processes on a semantical level. However, when semantic descriptions for the activities of process are available, we are able define correctness criteria for processes on the semantics level. Semantic validation should be interleaved with the actual modeling of the composed service: The user should be informed about problems with the composition in an unobtrusive way. Such problems, which can be seen as unresolved issues, arise from activities in the composition which violate one or more aspect of a set of desirable properties for well-formed workflows. According to [8], a composition is well-formed if

– one or more activities are contributing to the composition's overall end result,
– the inputs and preconditions of every activity are satisfied,
– every activity is either an end result or produces at least one output or effect that is required by another activity,
– it does not contain redundant activities.

Business Scenario As the last step, the modeler added the nearly invocable operation *Check Create Leave Request*. The tool highlights operations for which problems are tracked. As the added operation is not invocable, but nearly invocable, one input type is missing. The tool therefore marks the operation with a red border. This can also be seen in figure 2, where two out of four activities are highlighted. By clicking on the *Check Create Leave Request* operation, the user can open a panel showing its input and output types as inferred from the pre- and postconditions. The user sees that all input types of the operation are currently available in the composition, except *TimePointPeriod*, which is also highlighted using red color in this drill-down view. The user can also get an overview of all current problems with the composition by looking at the agenda.

The missing parameter *TimePointPeriod* represents the date or period for which the employee intends to request a leave. As our scenario has been taken from Duet, this data is provided by Microsoft Outlook after a the user selects a date from the calendar. In our example, the modeler therefore creates a human activity (modeling a task such as marking a period in the calendar) that produces a *TimePointPeriod* output. The modeler connects the human activity with the *Check Create Leave Request* operation. The coloring of the operation and the *TimePointPeriod* input type in the parameter view disappear and the issue is removed from the agenda.

Realization Information about operations with unsatisfied inputs can be retrieved by traversing the graph of the composition state. Listing 1.3 lists the pseudo code for aggregating all available types in the composition state. The available types are compared to the preconditions of each operation in the composition.

3.3 Suggest Partial Plans

Automated planners plan according to an algorithmic planning strategy, such as for example forward- or backward chaining of services. Human planners, in contrast, will not always behave according to this schema when modeling composed service. Users might have a clear idea about some specific activities that they want to have in the process, without a global understanding how the whole will fit together as a process. For example, they start modeling the composed service by adding some operations and chaining them together, and then continue with a non-invocable operation that is intended to be in a later stage of the composition. In such and similar cases, it is desirable for the user to let the editor generate valid service chains that connect two unrelated activities.

```
findUnsatisfiedInputs(State state, Role role)
2   for each operation in state do
      availableTypes = {};
4     availableTypes += role;
      requiredInputs = preconditions of current operation;
6     links = links connected to incoming plugs of operation;
      for each link in links do
8       recurseLink(link);
      if unsatisfiedInputs = requiredInputs − availableTypes != {}
10      store unsatisfiedInputs with current operation;

12 recurseLink(Link link)
    if link.source == start node
14    return;
    if link.source is a service operation do
16    currentOperation = link.source;
      availableTypes += currentOperation.postconditions;
18    links = links connected to incoming plugs of currentOperation;
      for each link in links do
20      recurseLink(link);
   else return;
```

Listing 1.3. Compute unsatisfied inputs in the service composition

Business Scenario In the last step the modeler resolved a problem with the *Check Create Leave Request* operation. If the user clicks on the operation to refresh the filtered list of available services, the tool will suggest the *Create Leave Request* operation. From the perspective of the user, this is the final operation. However, the modeler might not be familiar with the fact that a specific check operation needs to be invoked in order to create a leave request in the system. He then directly selected the *Create Leave Request* operation after the merge node depicted in figure 2. The modeler also creates the human activity producing the *TimePointPeriod* and links it to the *Create Leave Request* operation. Now, the modeler tries to create a link between the merge node of the parallel flow and *Create Leave Request*. The tool will detect that the set of postconditions up to the merge node does not satisfy the preconditions of *Create Leave Request* (the type *CheckCreateLeaveRequestResult* is missing). The tool instantly queries the semi-automated composition engine which detects that the insertion of the *Check Create Leave Request* operation would satisfy this open information requirement. The user is prompted whether or not the *Check Create Leave Request* should be inserted. The modeler approves this suggestion and the composition is complete.

Realization The suggestion of partial plans can be mapped to an automated service composition task: given an initial state and a goal state, a composition is created that leads from the initial state to the goal state. If we only want to suggest a partial plan to connect two operations A and B, we can derive the

initial state from the postconditions A and its preceding services. The goal state then consists of the unfulfilled preconditions of B. An approach from automated service composition is used to create a partial plan. For this purpose, we are using an extended version of enforced hill-climbing as presented in [4]. The principle of this algorithm will be briefly described in the following.

Enforced hill-climbing is a heuristic forward search algorithm in state space. Guided by a goal distance heuristic, it starts with the initial state and consecutively selects new services and reaches new states through the virtual invocation of selected services until the goal state is reached. Given a state all invocable services are determined. We use *findInvocableServices* mentioned above but without ranking. From these services, states are calculated using virtual invocation: Only the postconditions are applied without actually invoking the service.

For these states, goal distance estimations are calculated using a heuristic. The first state that has a lower goal distance estimation than the current state is selected as the new current state. If this state satisfies the goal state, we have found a valid composition. Otherwise, we continue until we have reached the goal state. In [4] we extended this algorithm to deal with uncertainty and to compose parallel control flows.

4 Related Work

In the following, we give a brief overview of related work in the field of semi-automated composition.

Sirin, Parsia and Hendler [9] present Web Service Composer. Their tool allows creating executable compositions of Web services that are semantically specified with OWL-S. In order to create a composed service, the user follows a backward chaining approach. He begins with selecting a Web service that has an output producing the desired result of the composition from a list of available services. Next, the user interface presents additional lists connected to each OWL input type of the service producing the end result. In contrast to the first composition step, these lists do not contain all available services: They contain only those services that generate an output satisfying the particular input type they are connected to. As a consequence, the plans constructed with the tool are not always optimal. For example, when one service operation delivers two outputs each of which satisfies a different input of a downstream service, this service operation occurs twice in the composed service.

IRS-III [10] includes a tool that supports a user-guided interactive composition approach by recommending component Web services according to the composition context. Their approach uses the Web Services Modeling Ontology (WSMO) as the language to describe the functionality of the Web services. Similar to Web Service Composer, the available services are filtered at each composition step. It is not possible to further shorten the filtered list based on nonfunctional properties. Our approach, in contrast, interprets the intended role of a service in form of a nonfunctional property when the weightings are assigned.

Kim, Spraragen and Gil introduce CAT (Composition Analysis Tool) [8]. At each composition step, CAT provides a list of suggestions about what to do next. These suggestions resolve errors and warnings, which are also displayed. The idea is that consequently applying suggestions will produce a "well-formed" workflow as a result. The authors therefore introduce a set of properties that must be satisfied by all operations in the composition in order for the process to be well-formed. The tool does not provide filtering functionality nor the ability to create partial plans.

PASSAT (Plan-Authoring System based on Sketches, Advice, and Templates) [11] is an interactive tool for constructing plans. Similar to CAT, PASSAT is not directly concerned with the creation of composed services, but its concepts can be mapped into the context of service composition. PASSAT is based on hierarchical task networks (HTN) [12]. In HTN planning, a task network is a set of tasks (or service calls) that have to be carried out as well as constraints on the ordering of these tasks. The HTN based approach imposes top-down plan refinement as the planning strategy the user must adhere to: The user can start by adding tasks to a plan and refine them by applying matching HTN templates. A template consists of a set of subtasks that replace the task being refined, as well as the postconditions of applying individual tasks and the entire template.

5 Conclusion

In this paper we presented the realization of three mixed initiative features for semi-automated service composition. We validated it in an implementation of a service orchestration tool, as well as case study and a service repository from a large vendor of service oriented business software. *Filter inappropriate services* allows for selecting only invocable services and performs a ranking among them. *Check validity* allows to check for the semantic correctness of a service composition. And, finally, with *suggest partial plans* we used an approach from automated composition to fill in gaps in service compositions. Our approach is currently the only one implementing all three mixed initiative features. The scenario shows that the ability to interleave all three features is very valuable for the modeler. Also, when service compositions are to be created by users without a strong technical background, usability plays a vital role. Our tool therefore does not impose a specific planning strategy on the modeler.

In the future, we plan to further validate the usefulness of our approach with end users. The goal is to show that these three mixed initiative features significantly improve modeling quality and reduce modeling time. In our current implementation we are facing performance issues when computing the list of nearly executable services. We are currently working on a realization strategy incorporating the partitioning of the ontology and distributing the computation. Finally, the area of semantic correctness is, in contrast to the syntactic correctness of service compositions and processes, still an open field: So far, our orchestration tool only covers unsatisfied inputs. Efforts to automatically discover redundant services are currently underway.

References

1. Zeng, L., Benatallah, B., Lei, H., Ngu, A.H.H., Flaxer, D., Chang, H.: Flexible Composition of Enterprise Web Services. Electronic Markets **13** (2003)
2. Pistore, M., Barbon, F., Bertoli, P., Shaparau, D., Traverso, P.: Planning and Monitoring Web Service Composition. Lecture Notes in Computer Science **3192** (2004) 106–115
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. Meyer, H., Weske, M.: Automated Service Composition using Heuristic Search. In Dustdar, S., Fiadeiro, J.L., Sheth, A., eds.: Proceedings of the Fourth International Conference on Business Process Management (BPM 2006). Volume 4102 of Lecture Notes In Computer Science., Heidelberg, Springer (2006) 81–96
5. White, S.A.: Business Process Modeling Notation, Working Draft (1.0). Technical report, The Business Process Modeling Initiative (2003)
6. Schaffner, J., Meyer, H.: Mixed Initiative Use Cases For Semi-Automated Service Composition: A Survey. In: Proceedings of the International Workshop on Service Oriented Software Engineering (IW-SOSE'06), located at ICSE 2006, 27–28 May, 2006, Shanghai, China, ACM Press, New York, NY, USA (2006)
7. van der Aalst, W.M.: Verification of Workflow Nets. In: ICATPN '97: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, London, UK, Springer-Verlag (1997) 407–426
8. Kim, J., Spraragen, M., Gil, Y.: An Intelligent Assistant for Interactive Workflow Composition. In: IUI '04: Proceedings of the 9th international conference on Intelligent user interface, New York, NY, USA, ACM Press (2004) 125–131
9. Sirin, E., Parsia, B., Hendler, J.: Filtering and Selecting Semantic Web Services with Interactive Composition Techniques. IEEE Intelligent Systems **19** (2004) 42–49
10. Hakimpour, F., Sell, D., Cabral, L., Domingue, J., Motta, E.: Semantic Web Service Composition in IRS-III: The Structured Approach. In: 7th IEEE International Conference on E-Commerce Technology (CEC 2005), 19-22 July 2005, München, Germany, IEEE Computer Society (2005) 484–487
11. Myers, K.L., et al.: PASSAT: A User-centric Planning Framework. In: Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, Houston, TX, USA, AAAI (2002)
12. Tate, A.: Generating Project Networks. In: Proceedings of the Fifth Joint Conference on Artificial Intelligence, Cambridge, MA, USA, Morgan Kaufmann Publishers (1977) 888–893