Internet collaboration and service composition as a loose form of teamwork

Padgham, Lin; Liu, Wei

Padgham, L., & Liu, W. (2007). Internet collaboration and service composition as a loose form of teamwork. Journal of Network and Computer Applications, 30, 1116–1135.
https://doi.org/10.1016/j.jnca.2006.04.006
Document Version: Accepted Manuscript

Published Version: https://doi.org/10.1016/j.jnca.2006.04.006

Internet collaboration and service composition as a loose form of teamwork

Lin Padgham[a] *, Wei Liu[b]

[a]School of Computer Science and Information Technology,
RMIT University, GPO Box 2476V, Melbourne 3000, VIC, Australia

[b]School of Computer Science and Software Engineering,
The University of Western Australia, 35 Stirling Highway, Crawley 6009, WA, Australia

*Corresponding author. Email Address: linpa@cs.rmit.edu.au

This paper describes Web Service composition as a form of team work, where the Web services are team members in a loose collaboration. We argue that newer hierarchical team work models are more appropriate for Web service composition than the traditional models involving joint beliefs and joint intentions. We describe our system for developing and executing Web service compositions as team plans in JACK Teams,^{TM}[1] and discuss the relationships between this approach and service orchestration languages such as Business Process Execution Language for Web Services (BPEL4WS). We discuss briefly how the use of AI planning can also be incorporated into this model, and identify some of the research issues involved. Incorporating Web service compositions into a mature Belief Desire Intention (BDI) agent team framework allows for integration of Web services seamlessly into a powerful application execution paradigm that supports sophisticated reasoning.

1. Introduction

With the rise of the world wide Web there has been a growing interest in Internet based services which can flexibly work together to perform more complex tasks than that performed by any single service alone. The promise of the possibilities of the semantic Web as initiated by Berners-Lee et. al. in 2001 [1], has created a vision of an enormous resource which is able to flexibly and robustly inter-operate to support enterprize level business collaborations.

Standards such as Web Services Description Language (WSDL) [2] and Simple Object Access Protocol (SOAP) [3] are developed to support uniform service descriptions and structured message exchange which are fundamental for inter-operability. However, service descriptions and message exchange alone do not themselves enable the integrated execution of stand alone services. We need also mechanisms for representing and developing the specification of what parts the individual services will play in the larger whole, and mechanisms for coordinating the execution of such a specification. Standards such as Business Process Execution Language for Web Services (BPEL4WS) [4], Web Service Flow Language (WSFL) [5] and more recently Web Ontology Language for Services (OWL-S) [6] start to address these issues at both syntactic and semantic levels.

One way of viewing service composition is as a loose form of collaborative teamwork [7]. Some people might argue that this is not really teamwork, as the members have no awareness of being part of a team or joint venture. However we would argue that by advertising services, the agents are in essence indicating a willingness to participate in internet teams. We suggest that a composite service can be seen as a team plan, which is executed by some collaborating group. In this paper, we take this approach, and describe Web service composition as a form of teamwork. We use an existing, powerful, BDI agent team development environment, JACK Teams^{TM}[8] as the basis for this work.

We describe a system which allows Web services to be incorporated into a JACK Team, or alternatively allows a composition of Web services to be the team. We execute the composition using the existing JACK Teams execution engine. This enables Web services to be integrated into a powerful agent reasoning system if desired. Possible failure or disappearance of a particular Web service (i.e. team member) is always an issue in open systems. We describe a failure

[1]JACK Teams^{TM} is the trade mark of an agent oriented team work model developed by *Agent Oriented Software Group*. A free evaluation package for JACK Intelligent Agents, which includes JACK Teams, is available for download from *www.agent-software.com.au*.

recovery algorithm that we have developed within our system to address such problems.

In our system, service compositions are developed interactively, allowing automatic code generation with minimal developer/user effort. We also describe how team plans and the necessary supporting code could be generated from BPEL4WS specifications. This would allow incorporation of existing specifications of composite services into a more complex agent based application.

Generation of Web service compositions using planning techniques is also an active area of current research (e.g. [9–11]). We explore how this might be integrated into our system, and report on some preliminary work done in this direction.

In the following sections, we first describe JACK Teams, the hierarchical teamwork framework that we use. Then we discuss the mapping of JACK Teams concepts to Web service concepts found in WSDL and BPEL4WS. Following this we describe the architecture and implementation of our system, including the failure recovery algorithm that we have developed. Section 5 describes different methods for developing the team plan or composition plan, including pointers to future work. The paper concludes with a brief discussion of the advantages of the team-based approach in service composition.

2. Agent Team Framework

The view of teamwork as proposed by Tambe and developed in STEAM [12], Teamcore [13] and most recently Machinetta [14] requires that team members are all operating with the same *joint intentions*. Implementations require that each team member has the same set of team plans, as well as needing to share all relevant beliefs with respect to the plan. Such a model of teamwork would clearly not apply to the kind of collaboration we may expect in open systems such as the Internet. In such systems, agents may well collaborate, and can even be seen to be operating as a team, in that their collaborative behaviors are appropriately coordinated. However they do not have shared goals or joint intentions, and it would be overly restrictive to expect them to share the same team goals. Rather they coordinate and collaborate for mutual benefit, but each has its own goals.

An alternative model of teamwork is developed within JACK Teams^{TM} which instead of requiring shared goals and intentions amongst members, reifies the *team entity*. It is this team entity that holds the team goal and executes the team plans. The team entity then coordinates the team members in doing their parts to achieve the team goal. The model is hierarchical, so team members may themselves be teams.

Unlike the STEAM model, JACK Teams, with some modifications, is quite suitable for the kind of collaborations that involve the use of a variety of Web services to achieve some more complex goals than can be achieved alone. In fact, team plans within JACK Teams provide a very powerful mechanism with reasoning capabilities for composing Web services, and then executing the composition in a coordinated manner.

In this section we describe first the various constructs required by JACK Teams, and then explore in detail how this maps to Web service composition in section 3.

2.1. JACK Teams Concepts

Jack Intelligent Agents^{TM} is an extension of the Java programming language, which provides agent oriented programming constructs for developing agent applications. It also provides a goal oriented execution engine which persists in trying all possible ways to achieve a goal, choosing the method most suitable at the current situation. It is based on the Beliefs, Desires, and Intentions (BDI) model [15]. The BDI agent model is an event-driven execution model providing both reactive and proactive behavior. In this model, an agent has certain beliefs about the environment, has goals (desires) to achieve, and has plans (intentions) describing how to achieve goals. JACK is one of a family of implemented BDI systems which include PRS [16], JAM [17], dMars [18] and Jadex [19]. JACK Teams is an extension of JACK Intelligent Agents^{TM} which provides constructs and support for *Team Oriented Programming* In JACK Teams a team is a distinct entity with its own representation. It incorporates the standard BDI reasoning mechanisms of JACK and other similar systems, with respect to behaviors such as choice of plans and persistence of goals if a particular plan fails. The team is in fact the core entity in JACK teams and an individual agent is simply represented as a team with no team members. We describe here some of the key concepts in the team model implemented by JACK Teams[2].

Team
Teams are characterized at the highest level by the *roles* they can perform, and the roles they require their *team members* to perform. They also contain a set of *team plans* for doing tasks related to achieving specific *goals*, or reacting to specific *events*. A team has a set of members which are (or can be) in a long term relationship to the team. In JACK Teams the team members are specified as belonging to a *role container*. Team members may be added and removed dynamically. These members can be assigned to (or requested to participate in) particular tasks (via JACK's *task teams*), according to the role(s) they can perform, and the roles required by a task as defined in the team plan.

Team Members
Team members can be either teams - sometimes called sub-teams - or individuals. An individual is represented in JACK as a team that does not contain any members and does not require any roles. As team members can themselves be teams, a team can be a hierarchical (or more complex) structure.

Roles
A role specifies the part that a member plays, or can play, within a team. It is defined in part by the goals for which that role is able to be responsible (or equivalently the tasks which it can achieve, or the events which it can respond to). In JACK Teams the *beliefs* or knowledge of the agent required for the role are also specified as part of the role. JACK Teams also associates a role with the events or goals generated by that role.

Team Plans
Team plans are a set of steps specifying how a task is to be achieved by members performing particular roles. Before a team plan can be executed[3] it must be established which team members, in which roles, will participate in this particular task. JACK Teams provides an *es-*

[2]The source of information for the following descriptions is primarily the JACK Teams manual [8], although in some places the concepts and terminology are based on our own experience and frameworks, which we then map to what is specifi ed for JACK Teams.
[3]It is also possible to delay establishing team members until such time as they are needed. However we do not consider that in this paper.

tablishment method which can be customized if desired. This method assigns team members that can perform the roles required within the plan. This sub-group is called a *task team* within JACK Teams.

Steps in the team plan are assigned to task team members via the roles as used within the team plan. JACK Teams provides a construct to allow members to perform steps in parallel if desired.

As with standard JACK plans, additional Java code can be incorporated within the team plan if necessary. Team plans (like standard JACK plans) are associated with a single goal to be achieved, event to be reacted to, or *message* to be responded to.

Goals, Events and Messages

Goals and events to some extent capture respectively the proactive and reactive character of agents (and teams). Messages capture the communication between agents (or teams which are not related to each other in the team hierarchy) which also requires some reaction or response. In JACK Teams (as in JACK) these are all represented by a similar data structure (Event and its subclasses) which contains arbitrary fields, and can thus be used for passing whatever information is needed beyond the particular goal/event/message type.

Beliefs

Beliefs in agent systems generally refer to all information the agent has about both its environment and its own state. Joint beliefs, as mentioned earlier are the beliefs held by all members of a team. Joint beliefs are not particularly important or supported in JACK Teams and its underlying model of teamwork, although they can be realized by belief propagation both up and down the team hierarchy. JACK, and also JACK Teams provides a specialized data structure called a *beliefset* which is represented and can be accessed in similar ways to relations in a relational database. JACK Teams allows specification of how beliefs are to be propagated between a team and its members.

2.2. JACK Teams Plan Execution

When a team decides to execute a particular team plan, the first step is to establish which team members will participate in the team plan. JACK Teams calls this *establishing the task team*. This is done by assigning team members from the relevant role containers, to each required role within the plan. An establishment method can be defined to choose amongst the members within a role container. Additional members can also be added to the team dynamically, in order to allow them to be used for the particular task.

Once the relevant team members have been identified for the particular task, the team plan can start execution. Steps within the plan request members to achieve particular goals. Requests are essentially messages containing the goal data structure, which has fields that can contain information relevant to the goal. This can also be used to pass back relevant information once the goal is achieved. Steps complete by either succeeding, in which case execution proceeds, or failing, in which case execution terminates, and a fail plan is executed. Failure of any step in a plan causes the plan to fail, at which point a new plan is searched for to achieve the same goal.

When a team member receives a request to achieve a goal it uses its own reasoning processes to determine how to achieve that goal - including using its own team members to delegate to. The team entity is not concerned with how the member carries out its responsibilities.

The control flow available in JACK Team plans includes the standard Java sequential, selective and repetitive constructs, plus also a parallel block. The parallel block allows various nuances. An `AND` variant requires all branches to succeed, whereas an `OR` variant requires only one. There are also variations regarding whether branches are terminated if a sibling branch succeeds or fails, as well as exception handling details.

3. Mapping JACK Teams to Service Composition

In this section, we will describe how JACK Team plans can use Web services as team members, and in fact, how team plans can provide a powerful mechanism for service composition.

In the commercial world, a composite service is either represented as *Orchestration* or *Choreography* [20]. Service orchestration specifies how services can interact at the message level to achieve some higher-level tasks. It describes the business logic and execution order of the interactions as abstract processes controlled by a single party. The abstract process becomes executable once bindings are made to the required individual service providers. Service choreography instead is used to track the messaging between multiple parties and assumes no one single party who “owns” the conversation.

Our interest in service composition is in how to fulfill tasks that cannot be achieved by a single service provider. Consequently our focus is on orchestration rather than choreography. When we refer to service composition, it is the orchestration aspect to which we refer. This implicitly assumes a *central controller* who executes the composite service according to the specification. This implicit single party (controller) has no clear identity, it can be the service consumer or a broker who provides composite services to end users.

3.1. Conceptual Mapping

JACK Team’s *explicitly defined* Team entity, which owns the team plan, maps well to the implicit controller of a composite service either as a consumer agent or a broker agent.

The team plan is then equivalent to the specifications found in service flow control languages such as WSFL [5] and BPEL4WS [20]. A similar facility is also provided by some service description languages (e.g. OWL-S [6]) in their provision of language constructs to specify composite services based on atomic services and processes.

In BPEL4WS composite services are described in terms of the actions of *partners* who take the delegation and carry out lower level business procedures. Service providers can also delegate tasks to other providers if necessary. These partners are conceptually equivalent to team members. The multi-level structure available via BPEL4WS (and other similar approaches), provides for similar structures to team hierarchies in JACK Teams.

The core of a WSDL description used by BPEL4WS is the *operation*. It describes the input and output *message*s of an abstract action. Information about which conversation protocol (e.g. *request-response* or *notification*) to use is implied by the presence and ordering of input and output. In JACK Teams, rather than an operation specifying the action requested of the Web service we have the goal that is to be achieved. If we wish to use an available Web service as a team member to achieve that goal, then we must map the goal to the appropriate WSDL operation. Goals (in JACK Teams) are represented by a data structure (JACK `Event`) with arbitrary attribute fields. Information required to achieve the goal (input parameters) and information needing to be passed back, resulting from achieving the goal, are carried in attribute fields of the goal (JACK `Event`) structure. The implementation then requires mapping of the information exchange to the expected messaging and conversation protocols in JACK statements.

3.2. Coordination and Control Flow

Typically, a flow control needs to define the message flow from the controller point of view. This may include, at the individual interaction level, initiating a request response process, waiting for certain matching message or sending a message in response to a received message. This is largely controlled by the individual service provider's service interface. In BPEL4WS, elements `<invoke>`, `<receive>`, and `<reply>` indicate the above communicative behaviors.

At the coordination level, the flow control needs to specify the execution order of the subtasks, be it sequential, repetitive, selective or parallel. In BPEL4WS, elements `<sequence>`, `<while>`, `<switch>` and `<flow>` describe these flow control respectively. Furthermore, the parallel control needs to determine the fork condition and join condition. A simple join condition in BPEL4WS is when all branches complete.

Obviously the specific constructs provided differ somewhat, but JACK Teams provides similar functionality to BPEL4WS. JACK tends to provide somewhat richer constructs, such as the join condition when executing branches in parallel, which has nuances surrounding the success or failure of particular branches, and the interpretation and management of this for the parallel block. For example it is possible to specify a parallel block which terminates with success as soon as one branch succeeds, then terminating other attempts.

At both the coordination and individual interaction level, the ability to wait for a certain period to allow synchronous communication is also fairly important. There should also be *timeout* conditions to recover from dead branches or processes. These are all supported in JACK Teams via constructs such as `@wait_for(trigger_conditions)`. The trigger condition can be used to implement a time out monitoring when a duration has expired, or some action has been completed.

4. Architecture and Implementation of Team Based Service Composition

Figure 1 illustrates the architecture whereby JACK teams can incorporate Web services as team members, also allowing JACK Teams to be the mechanism for describing and executing composite services.

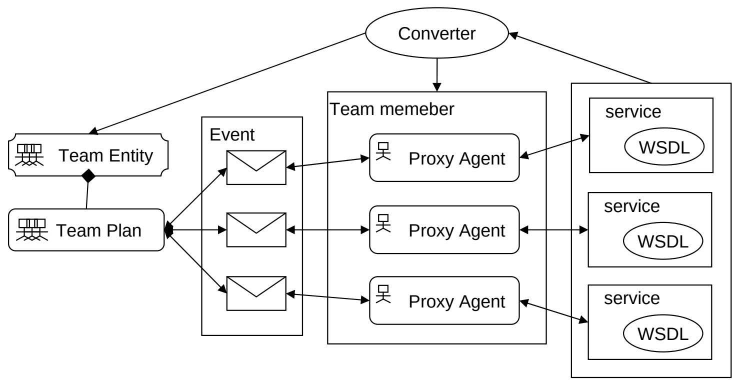

Figure 1. System Architecture

There are two separate stages supported within this system: team plan development, and team plan execution. During development, support is provided to generate JACK Teams code which will be suitable for use with the required Web services. This could be done manually by a programmer, but we have developed the support as part of an effort to make composite Web services widely available. This support is provided by the *Converter* module. The converter module also generates code which builds the *Proxy Agents* which at runtime provide the interface between JACK Teams and the Web services. These Proxy Agents are declared as being team members able to take on the appropriate roles (by virtue of being declared within the relevant role container in JACK Teams). We provide below further details about these two stages.

4.1. Development of Team Plans and Related Code

Our system does not currently have a module for locating and classifying service providers. This must currently be done manually, using facilities such as UDDI registries [21] or other mechanisms. The developer must choose which Web services to incorporate as (potential) team members. Having made this decision, the Converter module then produces the necessary JACK Teams code, which includes specification of relevant events, development of a proxy agent, inclusion of the proxy agent into the team, and appropriate requests to the proxy agent, from the team plan. The Convertor also produces the code which allows the proxy agent to communicate with the relevant Web service. If there are a number of Web services which conceptually offer the same service, then the respective Proxy Agents must be of the same type. This introduces some additional complexities into the event, which we do not go into here.

Once a service is selected from the service repository, by parsing the WSDL file, several programming entities are generated, including:

- a Proxy Agent as a representative for each Web service
- a Plan (or Plans) for the Proxy Agent, that can handle the relevant events, and communicate appropriately with the Web service
- the Role type that the Proxy Agent can play
- an Event that holds the input and output data for the service invocation, which is passed between the containing Team, and the Proxy Agent as a team member.

In JACK Teams a Role type specifies what Events a role performer can handle. It aggregates related Events into a Role specification class. An Event is responsible for passing information and triggering handling plans to process the information. So to a large extent, Events are similar to WSDL operations. The converter parses WSDL files and creates the Event specification necessary to carry all the information later needed by the Proxy Agent to request the particular Web service operation that is desired. From the JACK Team point of view, these Events are the subgoals it will be asking its team members to achieve.

WSDL's Port Type, as an aggregation of related operations, is comparable to a Role type. Port type in WSDL is uniquely identified by its `name` attribute. So in order to automatically generate the Role type class from WSDL, we use the Port type name as the class name for a Role type. The operations the Port type describes are then converted to different kinds of Events. The operation names are used as Event names. In the case of method overloading, where more than one operation shares the same name, the JACK Teams notion of multiple *posting methods* is used in the Event file to allow constructing the same type of Event using different data types. The input and output messages in an operation specification become the data members carried by the Event. An automatically generated Event file, can be seen in figure 2.

```
public event QueryProducts extends BDIGoalEvent{
    String port;
    Input in;
    Output out;
    #posted as postingMethod(String port, Input in, Output out){
        this.port = port;
        this.in = in;
        this.out = out;
    }
}
```

Figure 2. From WSDL Operation to JACK Event

Input and Output are classes that are generated according to the `<message>` element and the associated `<type>` element for each input and output element inside an operation. The `<binding>` information is also included in the corresponding classes.

A proxy agent is declared for each Web service, and it is declared to be able to fill the Role type related to that Web service. As a Web service client for the service provider, the proxy agent should be able to interact with the Web service using supported messaging protocols, typically including SOAP, HTTP GET, HTTP POST and MIME. In describing generation of the JACK Teams plan code we use SOAP as the example transport protocol.

The convertor generates plans for the Proxy Agent to handle each event type, which we recall are mapped to WSDL operations. In the plan body, the event (such as the `QueryProducts` event shown above) instance can be accessed to obtain the Input data structure and the port address, so a SOAP message can be automatically generated and sent to the service provider. For request/reply operations, a simple `@wait_for()` JACK construct is used to specify the wait condition and the time out alarm. When the SOAP message arrives back, it is parsed to create an Output object which is then stored in the Event, and the plan concludes. At this point the containing team is able to access the Event, and the newly obtained Output information, which can then be used, if desired, as Input for the subsequent service invocations.

4.2. Executing Team Plan with Web Services as Role Fillers

Once a Team Plan and supporting code has been built it must be compiled using JACK Teams. The program may consist of a single Team Plan representing simply a composition of Web services, or it may be a more complex application in which Web services are at times used as team members. The ability to integrate Web services into a larger BDI agent based application provides a powerful and flexible reasoning mechanism. However we concentrate in this description on the case of a single team plan representing a composition of Web services.

At run-time, once the decision is made to execute the specific team plan, the first decision is which team members to use for execution. In the simplest case, this is just a matter of choosing a set of proxy agents, representing particular Web services, which are able to fill the required roles. (Recall that JACK Teams provides an establishment method whereby code can be written to guide the choice of which members to fill roles if desired).

Once team members are assigned to roles, JACK Teams begins to execute the plan steps, which involve requesting proxy agents to achieve particular goals. The goal may include (in the event data) specific information related to this goal instance. For example, the goal may be to book a hotel, but the goal instance needs to include at least the information regarding dates and location. This goal request is passed to the proxy agent, and triggers the relevant plan for handling the particular goal (type). This plan then generates the specific Web service request, using the information contained within the event data. If a reply is expected the plan will wait for the reply, and then store the response into the output data of the event. Once the plan completes, control returns to the team plan of the containing team, and the output data is accessed.

In some cases the Web service may be offline, or may not respond as desired, or may not respond at all. In these cases the proxy agent will not be able to achieve the desired outcome requested of it as a team member. In this case it ends the execution with a fail. This returns control to the team plan. The default behavior in this case is that the team plan also fails. However, we have implemented some failure recovery methods to improve robustness, based on the assumption that in an open environment, there will often be alternative providers of services which can be used if needed, to prevent failure of the overall goal.

4.3. Failure Recovery from Within a Team Plan

We do not want the whole team plan to fail, simply because one member was unable to fulfill their assigned task(s), if there are other agents/services available to take over the failed agent's role. A major focus in Tambe's general model of teamwork [22] is improving the overall robustness of executing joint plans and a large portion of his teamwork model is concerned with the behaviors of failed team members. He has also done additional work in exploring the possibilities of monitoring the progress of team members in order to detect failure [23].

BDI systems provide powerful failure recovery mechanisms for individual agents, but these are not always the most appropriate for teams. When there are multiple team members available for filling a role in a given plan, then it is likely that one would want to try replacing a failed team member, before trying a whole new plan which would potentially waste a good deal of what had already been accomplished, and possibly assign a completely new task team. However, replacing a team member at the point of failure is not trivial, as it is necessary to take account of possible dependencies in earlier steps of the team plan. This may require some tasks to be redone in collaboration with the replacing member.

For example, imagine a situation where a user wishes to purchase a Hi-Fi unit for his office, and a CD, but also wants to check that the Hi-Fi unit chosen is one for which there is a suitable power supply amongst those available at the workplace. Assume there are agents selling Hi-Fi units, an information agent providing information about available power supply units and agents that sell CDs. It is necessary to select a Hi-Fi unit in order to check availability of the power supply, as this is different for different units. However the user does not want to commit to purchase until it is confirmed that an appropriate power supply is available. Figure 3 illustrates

Step	Role	Goal	Input	Output
1	HiFi Seller:	Provide quote	specification	HiFi model, price, quote num.
2	CD Seller:	Obtain CD	title, credit card num.	CD tracking num.
3	Power Supply:	Book power supply	HiFi model	booking num.
4	HiFi Seller:	Order HiFi	HiFi model, quote num., credit card	HiFi tracking num.

Figure 3. Hi-Fi Ordering Example

the interaction.

Assume that the first three steps execute successfully, but the fourth step fails as the agent has now sold out the Hi-Fi. We cannot simply replace the failed agent at this point in the plan. Rather we must go back to redo the first step. The third step will also need to be re-done, as it is dependent on the output of the first step. The second step on the other hand is independent and does not need to be repeated. Our system takes care of these dependencies, backtracking to redo only necessary steps.

The steps within the failure recovery algorithm are as follows:

1. Identify the failed step in the team plan,
2. Find an alternative team member (i.e. another proxy agent representing an alternative Web service),
3. Mark all the interfaces associated with the failed team member as *not executed*,
4. Identify any other steps which depend on the output of failed steps and mark as *not executed* (perform this recursively),
5. Re-execute the plan, skipping steps that are marked as executed.

To implement the failure recovery strategy, we implemented an additional data structure which keeps track of the current execution state of a team plan. The data structure holds information required to perform the failure recovery, which includes information such as an *execution-state flag* (true if it was successfully executed), *dependency with other steps* in the plan, and *assignment of a team member to a role*. Dependencies between the steps of the team plan are analyzed at compile time.

This approach is particularly suitable for open environments where it is quite likely that on occasions a service previously used or identified may be unavailable. Alternative services are typically independent of each other since they are deployed by different organizations, and consequently failure of one is likely to be independent from the failure of others. Although the approach was designed with open environments in mind, it can also be applied to any team environment. It ensures that once a team plan is chosen for execution, it will not be given up until all replacement options for failed agents have been tried. This provides an additional level of commitment and robustness, beyond the BDI agent commitment to trying different plans to achieve a goal.

5. Developing the Team Plan(s)

One way to develop a JACK Teams plan representing a service composition, is by programming it directly in JACK. However this also requires quite a bit of additional programming to develop the proxy agents. The approach we have taken is to develop an interactive interface which assists the user in developing the team plan using appropriate Web services. Other methods of producing the team plan and associated code include translation from alternative composition representations, or a planner which builds the composition by translating WSDL or OWL descriptions into planning operators and then converting the planner output to JACK Teams structures. We describe first the interactive approach we have implemented, and then describe how the other approaches would work, including some preliminary work that we have done there.

5.1. Development Using an Interactive Tool

In order for the Semantic Web to enjoy a similar level of acceptance as the current WWW, it seems that it will be necessary to assist users to access and compose Web services, in the same way as Web browsers and Web page development tools currently assist users in accessing and developing Web pages. We developed a prototype system known as **A**gent **S**ervice **K**omposition **I**nterface **T**ool (**ASKIT**) [4], to explore this vision [7].

Our prototype system was originally built for, and deployed in, the Agentcities network[5]. Agentcities is an international initiative, which aims at facilitating the exploration of a world-wide open network of platforms hosting a variety of inter-operable agent services The communication mechanisms were slightly different and the Agent Communication Language was significantly more sophisticated than SOAP, but the system is readily modifiable to use with Web Services described by WSDL descriptions.

ASKIT forms the basis of the architecture of our system as described in section 4. It consists of four main components:

- *Web interface* is responsible for collecting data from end users.
- *Service discovery and registry module* keeps service provider details, service types and service APIs. It is also responsible for populating the Web interface and displays available service APIs in an intuitive manner to the end user.
- *Team plan generation module* is responsible for generating abstract composite service specifications from the Web interface data, generating abstract team plans, backtracking upon plan failures, and generating *proxy agents* for interacting with service providers.
- *Team plan execution module* is responsible for executing the team plan, interacting with the required Web services. The team plan execution module is essentially the system which has been described in section 4.

[4]The prototype of ASKIT is available at http://agentcities.cs.rmit.edu.au:7198/agentcities/login.jsp.

[5]Agentcities (http://www.agentcities.org) grew out of the work of the Foundation for Intelligent Physical Agents (FIPA - http://www.fi pa.org), which is a standards body focussing on standards and infrastructure for rational interaction and cooperation among heterogenous agents.

All the Web services are organized and displayed to the user in terms of service types, with associated interfaces. Each interface has a list of expected input parameters and a list of expected output value types. For example, a bank service type has interfaces to open an account, it takes your name, address and passport number as input values, and it returns your account number as an output value of the interface

Multiple instances of the same type of service can be available on the network simultaneously, but the user need only specify the abstract service desired. This allows the system to bind at runtime to an appropriate service, depending on the configuration of the environment. This is important given the dynamic and evolving nature of the Web environment, where one cannot rely on particular services being available at any given time.

The interface also allows a user to specify the control sequence of a composite service by combining the existing services sequentially, or in parallel, or a combination of both, using simple menu selections. The team plan generation module uses the information generated at the user interface to build JACK team plans, which at the request of the user can be compiled and run, or saved for later use.

The service discovery and registry module is an area where significant work still remains to be done. Currently, the system only interacts with a known set of registries, i.e. directory facilitators for service discovery. Ideally this module would have sophisticated abilities to locate Web services, using UDDI directories and other repositories, and by trawling the WWW, locating and categorizing services found. Nevertheless, the limited nature of this module still allows the demonstration of interactive service composition.

The team at FOKUS[6] in the SATINE project[7] have also recently developed an interactive tool for specifying Web services using BPEL4WS and OWL-S [24].

5.2. Translation from Special Purpose Web Service Languages

A number of languages are developed for describing Web service composition, including Microsoft's XLANG, IBM's WSFL, or the later BPEL4WS, from IBM, Microsoft and BEA [4,20]. OWL-S process descriptions [6] are also in this category.

We will take one of these, BPEL4WS, and indicate how it can be transformed into a JACK Team plan and the necessary supporting code. In section 3.1, we looked briefly at the language constructs for BPEL4WS at the conceptual level. In this section, we will discuss in more detail the translation from a BPEL4WS process specification to a JACK plan.

If we say a WSDL file describes *what* services and possible collaborations between the services are available, a BPEL4WS file describes *how* the services and partnerships defined in one or more WSDL files can be scheduled and organized into some executable process that provides an integrated service. BPEL4WS assumes the service description files in WSDL define some `<partnerLinkType>`, which indicates peer interactions between partners or business collaborators. A partner link type defines one or more types of roles, and each role corresponds exactly to one port type in WSDL. For example see figure 4.

A partnerLinkType in WSDL specifies the kind of collaborations that are possible, and can be used in the BPEL4WS file for partnership verification. A composite service as a BPEL4WS business process first describes the instance partner links from the central controller's perspective. It specifies the capabilities of each partner and also the controller as `partnerRole` and

[6]The Fraunhoffer Institute for Open Communication Systems
[7]http://www.srdc.metu.edu.tr/Webpage/projects/satine/

```
<partnerLinkType name="BuyerSellerLink"
   xmlns="http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
  <role name="Buyer">
    <portType name="buy:BuyerPortType"/>
  </role>
  <role name="Seller">
    <portType name="sell:SellerPortType"/>
  </role>
</partnerLinkType>
```

Figure 4. Example of a partnerLinkType declaration in WSDL

`myRole` respectively. It then specifies how the collaboration process can be carried out step by step.

Using the XML Schema for BPEL4WS, we can generate a *document handler interface* to define the skeleton of a handling method for each element in BPEL4WS. The implementation of this interface can be used in a BPEL4WS document *parser* (which can also be generated based on the Schema), to define the exact behavior when an element is encountered when parsing a BPEL4WS process description.

A JACK Team Entity, a BDIGoalEvent and a team plan skeleton can be automatically generated. The team, team goal and team plan can all take on the business process name as their corresponding file name with appropriate extensions.

The data items carried through a BPEL4WS process are declared according to their WSDL message types. Once a `<variables>` tag is encountered, the `handleVariables` method in the document handler can parse the enclosed tags and the message tags in WSDL to create a class for each type of message. An instance variable of each message type can then be created in the team goal event file so that the event holds a list of data members of the specified message types.

A second block of parsing would be needed for the `<partnerlink>` tag. The `handlePartnerlink` method in the handler can look for the partner role and generate JACK team members. The role definition in the WSDL file contains port type then operations. As discussed previously (section 3.1), WSDL operations map to JACK events, and the plans that handle those events. Consequently a JACK Team member must be created with declarations regarding handling of the roles associated with the events corresponding to the operations. All the JACK Role types generated from the `partnerRole` specification must be declared in the Team entity using role requirement declarations(`#requires role`). BPEL4WS `myRole` is performed by the Team entity itself and instead of a role declaration requires a declaration that it handles the relevant event (`#handles>`.

Once this second block of parsing activity is completed, all necessary structures have been declared and mapped as shown in figure 5. What remains is to map the process steps.

The process steps in BPEL4WS consist of three basic *communicative acts* `<receive>`, `<reply>` and `<invoke>`. The control flow organizes the interaction into sequen-

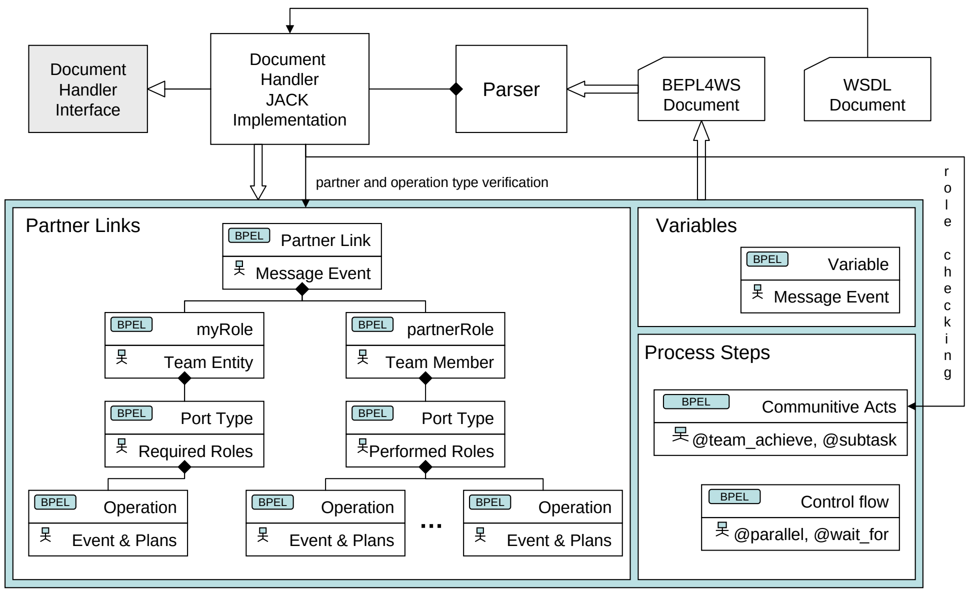

Figure 5. Creating JACK Teams using BPEL4WS

tial, parallel, repetitive, selection and triggered selection using `<sequence>`, `<flow>`, `<while>`, `<switch>` and `<pick>`. The data flow is simply indicated by `<assign>` element for copying data from one variable to another. The document handler can translate each of these elements into JACK Teams syntax to build the team plan body.

```
<receive partnerLink="purchasing"
         portType="lns:purchaseOrderPT"
         operation="sendPurchaseOrder"
         variable="PO">
</receive>
```

Figure 6. Example showing possible attributes of a receive communication in BPEL4WS

The three communicative acts use attribute/value pairs to indicate which partner performs which action using what variables as shown in figure 6. In JACK Teams, each of the three communicative acts is implemented either as a subtask of the team entity or a subtask delegated to a team member. As the role performer in the communication act specification is not explicit, some extra calculation is necessary. The `portType` specification can be used to find the `role` that encapsulates it in the WSDL file. In the BPEL4WS file, the `partnerLink` name and the `role` should uniquely identify *who* is performing the role, `myRole` or a `partnerRole`. `@subtask` is used for the task to be performed by `myRole`, while `@team_achieve()` is used for the task for `partnerRole`.

Sequencing, looping and selection in JACK teams use standard Java. Parallelism in JACK team plans is implemented using `@parallel()` which allows specification of a block of statements to be done in parallel. The `<pick>` construct in BPEL4WS allows you to block and wait for a suitable message to arrive or for a time-out alarm to go off. When one of these triggers occurs, the associated activity is performed and the pick completes. JACK's `@wait_for()` can be used together with it's *time cursors* and *action cursors* to implement this.

One of the main motivations for translating BPEL4WS (or similar composition descriptions) specifications into JACK Teams would be in order to integrate the composite service into a larger application supporting agent reasoning.

5.3. Service Composition Using a Planner

One automated way to generate collaborative team plans, is by using AI planning techniques. Examples of such systems are SWORD [25] which takes a rule-base approach, McIlraith et. al. [9] which uses Situation Calculus for planning (Golog) and Wu et. al. [11] which uses the hierarchical task-reduction planner SHOP2 as their service composition generator. Some recent work has even used Linear Logic [26] for this purpose.

We have begun to do some work using a combination of WSDL descriptions and the SHOP2 Hierarchical Task Network (HTN) planner. HTN planning is a planning technique relying upon a predefined set of task reduction schemata to reduce high-level abstract tasks into a suitable ordering of primitive subtasks. Figure 7 shows an example task reduction schemata for HTN planning.

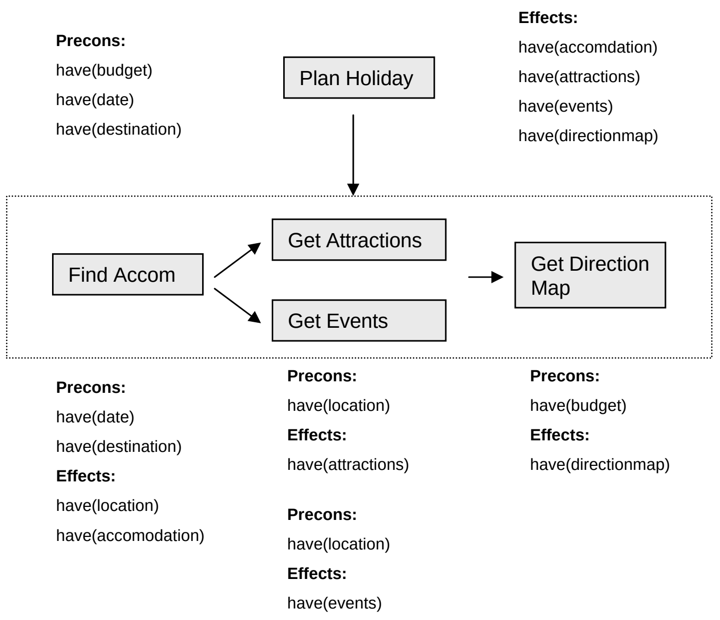

Figure 7. An Example Task Reduction Schemata

Like other traditional planning techniques, HTN planning implemented in SHOP2 considers actions as a state altering `operators`. Each operator is a four tuple, including a head defining the signature of the operator, the preconditions, the add list and the delete list in terms of world state. The description of a planning domain in SHOP2 therefore includes a set of operators, a set of literals that denote the state of the world, and also the task reduction schemata (i.e. `methods`) indicating how complex tasks can be reduced into primitive operators that can be directly executed.

Currently we have implemented a system which transforms WSDL descriptions into SHOP2 operators which can then be used to find a plan (to achieve some end state) which uses a number of services. Once the plan is found it is executed by invoking the relevant Web services. The translation from WSDL to SHOP2 is currently done manually.

The atomic processes in WSDL are described as `<operations>`, with input and output. When parsing a WSDL description for conversion to SHOP2, each operation is created as an operator in SHOP2, with WSDL operation input as parameters or preconditions and the output as a collection of effects. Because there is no process description in WSDL, there will be no SHOP methods generated. Therefore the planning is done at the primitive operator level. A set of WSDL files can be processed into a list of operators which preserve the port reference of WSDL operations by naming the operators as a concatenation of the port address and the operation name. The operator name can then be used later to identify the service provider and to resolve the issue of multiple service providers using the same operation name for different actions.

We do not currently map the generated plan into JACK Teams, but in principle, it can be used with the WSDL files to produce the necessary code, with each plan step mapping to a `@team_achieve()` statement.

Sirin et. al [10] have published work providing algorithms for converting OWL-S process descriptions into SHOP2 operators and methods. An atomic process in OWL-S is modeled as a SHOP2 operator, whereas simple processes and complex processes are translated into a SHOP2 method or collection of methods. This allows for better use of task reduction abilities within SHOP2 than the mapping directly from WSDL which we have been doing.

One difficulty in mapping between service descriptions and planner operations, is the fact that pre-conditions and post-conditions so crucial to planning do not really have a straightforward mapping in Web services. Also, the information typically provided as the output of the Web service does not have an explicit match in planning, although we can treat the output as knowledge effects. Sirin et. al [10] actually suggest that because information services do not alter the world state, they should not appear as a step in the final plan. However, a large amount of currently available services on the Web are information services, and we consider that a planning approach to service composition must take account of these, and of the information collection tasks necessary for successful overall task completion. Also, AI planning does not concern itself with the identity of those executing operations.

Although there are a number of problems to be resolved in order to successfully use planning as a mechanism for generating service compositions, it is an interesting area and one in which significant work is being done.

6. Discussion and Conclusion

In this paper we have presented a view of service composition as a loose form of teamwork. In particular we have shown how the JACK Teams development platform can incorporate Web services as team members. We have also shown how JACK Teams can be used as an infrastructure for supporting service composition. We have described the support we can provide to the developer for incorporating Web services into JACK Teams. We have also described the architecture of the system that, together with JACK Teams, provides the execution engine for the composite services developed. Finally we have described the failure recovery mechanism which we have developed which enables failing team members to be replaced, within a team plan, while ensuring that dependency constraints between members are maintained.

One advantage of our approach is that JACK Teams offers the full power of a programming language (Java plus JACK team constructs) for constructing service compositions. This means that additional computation, specific to a certain composition plan can easily be incorporated. Our system enables execution of service compositions in a Web services environment as JACK Team plans.

A major advantage of this approach is that a composite Web service is owned by a reasoning entity (the agent owning the team plan) and it is therefore possible to include a range of options beyond simply coordinating requests from a number of services. The fact that JACK Teams is a BDI system, means that service compositions can be integrated into a structure of multiple alternative plans for achieving a goal. If one method fails, the execution engine will automatically pursue alternatives.

A further advantage of using JACK Teams is that it has built in the notion of a team that endures beyond the lifetime of a particular collaborative task. This team can change dynamically during execution. Consequently it can readily be used for supporting a form of coalition formation or alliance building based on interaction experiences. Rather than initially having all available Web services be team members, a preferred set can be declared as team members, while others can be declared as available for incorporation if needed. If a particular Web service was successfully used (by incorporating it into the team) it would then remain a member of the team, and thus be more likely to be used in future. Similarly non-performing Web services could be dynamically discarded from the team. This provides a mechanism for evolving business coalitions or other sets of preferred providers.

There are some aspects of the JACK Teams platform which cannot be made available to Web services. For example, the belief propagation which allows efficient management of sharing beliefs between team members and the team to which they belong, cannot be used with members which are Web services. Nevertheless, JACK Teams offers a sophisticated level of support for the service composition task.

Currently we do not produce WSDL descriptions of composite services to allow them to also be advertised in their own right as Web services. This is clearly something that can be done, but is not something we have developed any automated support for at this stage. The area of service discovery, and incorporation of this into the system, both at design time and runtime is also an area needing further work. Our conclusions so far are that this is a promising approach to developing robust and flexible composite services. The concept of teamwork can include the kind of collaboration found between Web services, as long as the teamwork model used is not reliant on concepts such as joint beliefs, shared goals and joint intentions.

Acknowledgements

We would like to acknowledge the support of the *Australian Research Council* through grant #LP0218928, the *Australian Department of Education, Science and Technology*, through IAP grant #CG040014 which supports us to collaborate with EU project SATINE, and the *University of Western Australia* which provided a university research grant to the second author. We also acknowledge the ongoing support and collaboration with the *Agent Oriented Software Group*. Kenichi Yoshimura contributed significantly to aspects of this work, especially in development of the software infrastructure. We also thank many colleagues and students who have contributed to discussions on this work, and to testing the system.

REFERENCES

1. T. Berners-Lee, J. Hendler, O. Lassila, The semantic web, Scientific American.
2. E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, Web services description language (wsdl) 1.1 language specification, World Wide Web Consortium (W3C) Note (March 2001).
3. M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, H. F. Nielsen, Simple object access protocol (soap) version 1.2 specification, W3C Recommendation (June 2003).
4. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana, Business process execution language for web services - language specification, IBM Developerwork Language Specification (May 2003).
5. F. Leymann, Web services flow language (wsfl) specification version 1.0, IBM Software Group (May 2001).
6. A. Barstow, J. Hendler, M. Skall, J. Pollock, D. Martin, V. Marcatte, D. L. McGuinness, H. Yoshida, D. D. Roure, Owl web ontology language for services (owl-s), W3C Submissions (November 2004).
7. K. Yoshimura, L. Padgham, W. Liu, An infrastructure for agent collaboration in open environments, in: T. Gedeon, L. Fung (Eds.), AI 2003: Advances in Artificial Intelligence, Springer Verlag LNAI 2903, Perth, Australia, 2003, pp. 612–623.
8. Agent-Oriented Software Group, JACK Intelligent Agent Teams Manual, release 4.1 Edition, address: P.O. Box 639, Carlton South, Victoria, 3053, AUSTRALIA (May 2004).
9. S. McIlraith, T. C. Son, Adapting golog for composition of semantic web services, in: The Processings of the Eight International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France, 2002, pp. 482–496.
10. E. Sirin, B. Parsia, D. Wu, J. Hendler, D. Nau, Htn planning for web service composition using shop2, Journal of Web Semantics 1 (4).
11. D. Wu, E. Sirin, J. Hendler, D. Nau, B. Parsia, Automatic web services composition using shop2, in: Workshop on Planning for Web Services, Trento, Italy, 2003.
12. D. V. Pynadath, M. Tambe, Automated teamwork among heterogeneous software agents and humans, Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) 7 (2003) 71–100.
13. D. V. Pynadath, M. Tambe, The communicative multiagent team decision problem: Analyzing teamwork theories and models, Journal of Artificial Intelligence Research 16 (2002) 389–423.

14. P. Scerri, D. V. Pynadath, N. Schurr, A. Farinelli, S. Gandhe, M. Tambe, Team oriented programming and proxy agents: The next generation, in: Proceedings of 1st international workshop on Programming Multiagent Systems, Springer LNAI 3067, 2004.
15. M. E. Bratman, Intentions, Plans, and Practical Reason, Harvard University Press, Cambridge, MA, 1987.
16. M. P. Georgeff, A. L. Lansky, Procedural knowledge, Proceedings of the IEEE Special Issue on Knowledge Representation 74 (1986) 1383–1398.
17. M. J. Huber, JAM: a BDI-theoretic mobile agent architecture, in: Proceedings of the Third International Conference on Autonomous Agents (Agents'99), Seattle, USA, 1999, pp. 236–243.
18. M. d'Inverno, D. Kinny, M. Luck, M. Wooldridge, A formal specification of dMARS, in: M. P. Singh, A. S. Rao, M. Wooldridge (Eds.), Intelligent Agents IV: Proceedings of the Fourth International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag LNAI 1365, 1998, pp. 155–176.
19. L. Braubach, A. Pokahr, W. Lamersdorf, Jadex: A short overview, in: Proceedings of Net.ObjectDays: AgentExpo, 2004, pp. 195–207.
20. C. Peltz, Web services orchestration - a review of emerging technologies, tools and standards, Tech. rep., Hewlett Packard, Co (Janurary 2003).
21. O. for the Advancement of Structured Information Standards, The uddi white paper - introduction to uddi: Important features and functional concepts.
22. M. Tambe, W. Zhang, Towards flexible teamwork in persistent teams: extended report, Journal of Autonomous Agents and Multi-agent Systems, special issue on "Best of (ICMAS) 98".
23. G. A. Kaminka, M. Tambe, Robust agent teams via socially attentive monitoring, Journal of Artificial Intelligence Research (JAIR).
24. M. Flugge, D. Tourtchaninova, Ontology-derived activity components for composing travel web services, in: The International Workshop on Semantic Web Technologies in Electronic Business (SWEB2004), 2004.
25. S. R. Ponnekanti, A. Fox, Sword: A developer toolkit for web service composition, in: Eleventh World Wide Web Conference, Honolulu, HI, USA, 2002.
26. J. Rao, P. Kungas, M. Matskin, Application of linear logic to web service composition, in: First International Conference on Web Services, Las Vegas, USA, 2003, pp. 3–10.