A DYNAMIC AGGREGATION MECHANISM FOR AGENT-BASED SERVICES

Makram Bouzid
Jérôme Picault
David Bonnefoy

Motorola Labs
Parc les Algorithmes - Saint-Aubin, 91193 Gif-sur-Yvette, France

Keywords: Dynamic Service Aggregation, Agents, Multi-agent Systems, Constraint Satisfaction.

Abstract: At a time when the web is switching from a data-oriented view to a service-oriented view, we can envision an environment where services are dynamically and automatically combined to solve new problems that one single service cannot solve. Agent technology provides a good basis for creating such an environment but many issues remain to be solved. This paper presents a step towards a dynamic service aggregation mechanism, introducing a pragmatic approach and an implementation. This work was carried out in the context of the Agentcities.RTD project.

1 INTRODUCTION

The web is switching from a data-oriented view to a service-oriented view, through initiatives such as the Semantic Web (W3C, 2001a) or the Web Services (Mc Ilraith et al., 2001).

Pushing this vision further, we can envision a revolutionary way of accessing Internet services. The result of a service should seamlessly derive from users' wishes rather than from users' skills. The ultimate step is to break the limitations of existing Web services, to enable them to combine themselves in order to solve new problems that one single service cannot solve. Intelligent agents represent a promising technology to face these challenges, since they enable to hide much of the complexity of accessing Web services, while bringing additional value by customizing and composing the services.

A fully dynamic service aggregation mechanism would provide a lot of benefits:

- New services, potentially unanticipated during design time, can be constructed to address a specific problem, on demand of the system or its users;
- A relatively large number of services can be constructed from a set of basic service components;
- The human involvement can be minimized, as well as the system disruptions to perform upgrades and addition of new functionality.

This paper presents a first step towards such a dynamic service aggregation. This work was conducted in the context of the Agentcities project, and carried out in the same spirit: working both at the theoretical and practical levels. A first prototype of the system described here was implemented.

The paper is organized as follows. Part 2 describes the context of this work and defines more precisely what we mean by *dynamic service aggregation*. Part 3 gives an overview of our aggregation mechanism and a detailed description as well. In part 4 a first application of this mechanism, called Evening Organizer, is presented. We finally conclude with some future directions of that work.

2 CONTEXT

2.1 Agentcities

The Agentcities project (Agentcities, 2003) is developing a foundation for the vision of an ambient proactive environment where heterogeneous, autonomous and increasingly intelligent systems representing businesses, services and individuals are able to interact with each other and enable dynamic composition of services. Agentcities is creating a realistic, decentralized and open environment based on agent technology, which will enable high-level semantic interoperability between systems and build knowledge and understanding of dynamic open environments for eventual transition to usage in reliable commercial

Bouzid M., Picault J. and Bonnefoy D. (2004).
A DYNAMIC AGGREGATION MECHANISM FOR AGENT-BASED SERVICES.
In *Proceedings of the Sixth International Conference on Enterprise Information Systems*, pages 10-15
DOI: 10.5220/0002638900100015

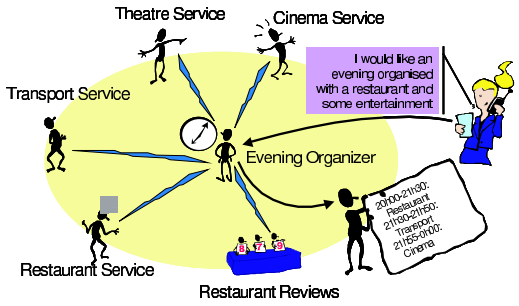

Figure 1: EO service aggregation

grade systems. Agentcities goes beyond Web Services by taking advantage of the interaction models, formal languages characteristics of agent technology. Platforms and services in the environment are publicly accessible and form a public resource for the testing, deployment and usage of dynamic services.

One of the demonstration scenarios of Agentcities is an *Evening Organizer (EO)*, which is an agent entity that enables dynamic composition of entertainment services such as cinemas, restaurants, etc. and integration with business services (Figure 1).

2.2 Service Aggregation and Composition

In the domain of services, many activities (research and standardization) aims at providing solutions for the description of services, the interoperability of services and the automation of their invocation and aggregation. We may distinguish three levels of ambition in those works.

The first level provides collaborating services with knowledge of their specification and interaction details at design-time. Web services with XML-based languages such as UDDI (UDDI, 2001) (repository of businesses) and WSDL (W3C, 2001b) (Web Service Description Language) represent those types of services. Service invocation and interoperability are possible by exchange of SOAP messages (Box et al., 2000). The specifications of such exchanges are detailed in the WSDL description of the service. However a computable part must still be hand-coded. A complex service can be described through WSFL (Web Service Workflow Language) (IBM, 2001) by a composition of primitive services. This description is made by hand and there is for the moment no proposition to perform dynamic service aggregation.

The second level provides collaborating services with knowledge of their specification details at design-time, but their interactions are defined at run-time. Those services are principally represented by agent services, which have flexible communication capabilities, but there are some non-agent proposals at this level, allowing a kind of dynamic composition of services. For example, the SWORD toolkit (Ponnekanti and Fox, 2002) allows doing such dynamic composition based on a combination of service inputs and outputs of elementary and final services as the principal constraints.

The third level provides collaborating services with knowledge of their specification and interaction details only at run-time. This is the highest level and the most interesting one, but the more complex to obtain as well.

While the terms *aggregation* and *composition* are usually used indifferently, we would like to make a distinction. Service composition is reserved for the case when services are chained one after the other, each service using as input the output of the previous service, as in a mathematical composition. Service aggregation is the general case when one service is built from several other services, though there may or may not be direct interaction between the different services. Services are combined into a new, composite service.

2.3 Scope of the work

In our work, we focused on service aggregation as described above, positioning our architecture between the second and third levels of automation. Indeed, services interactions are provided at run-time but in addition their specifications are also discovered at run-time (e.g. looking at a repository).

We have made some simplifying assumptions. We suppose the environment is composed of a large number of services, several potentially being interchangeable. The services that are aggregated do not interact with each other directly, even if what is found for a particular service may influence the choice of another service. In addition, the aggregation mechanism is centralized: though the ”execution” of the new compound service is distributed.

3 DYNAMIC AGGREGATION ARCHITECTURE

3.1 General considerations

The aggregation of agent services can be seen as a kind of distributed problem solving and planning, which necessitates communication, negotiation, and planning capabilities between agents to aggregate a service, as well as coordination abilities for its later execution. Several models for distributed problem solving through agents have been proposed in the lit-

erature, using either centralised or distributed planning methods for distributed or centralised plans generation (Weiss, 1999). But the proposed models are either too theoretical to be completely and successfully implemented, or implementation-oriented, like those proposed for cooperative robots, and cannot easily be generalised to be applied for agent services aggregation. HP proposed a framework called DySCo, consisting in a service model and a reference infrastructure for system implementation (Piccinelli and Mokrushin, 2001). This framework allows dynamic aggregation of e-services (not agent-based services), using the concept of service incompleteness, in terms of implementation, and multiparty orchestration.

We decided to adopt a more pragmatic approach, keeping the notion of service aggregation as a kind of planning problem.

3.2 Overview of the aggregation process

In our model, an aggregated service is represented by a plan that at first will be partial and non-instantiated, and gradually refined until a complete and instantiated plan is obtained.

Our approach is similar to the one described by Wooldridge and Jennings in (Wooldridge and Jennings, 1999) for cooperative problem solving, in terms of process stages, but we focus here on the *team formation* and *plan formation* phases.

At a high level view, our aggregation mechanism is thus composed of the following five stages:

1. Understanding the definition of the new service and design of its structure. According to the taxonomy of services, their policies and the interrelationships among them, this definition may be useful to limit the scope of the services needed, or to build the new service skeleton. This will allow building e.g. a parallel structure for an emergency service that interacts with firemen and police services at the same time and a sequential structure for a succession of entertainment services;
2. Creation of a partial and non-instantiated plan (i.e. a skeleton) that is expected to solve the problem. This plan is built from the user's requirements and the definition of the new service;
3. Finding the types of services that will compose the final service; this requires some reasoning capabilities;
4. Finding and selecting the associated service providers, i.e. agents delivering the services, according to pre-defined constraints and policies;
5. Finding the right combination of service instances, supplied by service providers, allowing to build a complete and coherent plan while respecting all the user constraints. This is a constraint satisfaction problem, which we can solve by using constraint programming techniques, such those proposed in the artificial intelligence (AI) or operational research (OR) domains. The algorithms should be chosen according to the application type and domain, and depending on the user request, we can either return all the solutions or the optimal one(s).

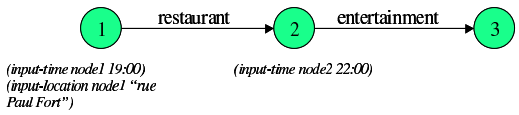

Figure 2: Non-instantiated plan

Our aggregation mechanism is dynamic since:

- Searching the agents offering a given service is performed on the fly;
- Some services may be dynamically inferred and inserted to a plan instance by the aggregating entity (i.e. a special service performed by an agent) without being asked for by the user;
- New services and new types of services can be added dynamically too without interrupting the existent system.

We detail in the following section the five steps of our aggregation mechanism.

3.3 Aggregation process details

The first step consists in reasoning about the user requested compound service. This service will consist in a set of elementary services. This set is currently provided by the service requester (i.e. a person, another service or an agent).

The second step concerns the partial plan creation that we will describe in details in the following section given its importance to the aggregation process.

Plan Description

A plan is an ordered set of event resource objects that are restricted by appropriate constraints on their properties and the interrelationships among such properties. A plan is represented by a *precedence graph*. If we refer to the evening organizer (EO) example, a plan will contain the services the user will use during an evening (Figure 2).

According to (Blum and Furst, 1997), a planning graph allows explicitly encoding all the planning problem constraints, reducing the amount of search needs, and permitting to construct plans quickly. We adopted a similar representation of plans using a graph for aggregated services. It allows encoding the

planning constraints and backtracking too, but plans are managed in a slightly different manner.

Each node represents the user state at a given moment during the execution of the aggregated service, e.g. in the EO before and after having a dinner states. These states are represented by a set of constraints. Each *edge* is labelled with an event. An event represents a *type and/or an instance of a service resource* (e.g. "Restaurant", "Entertainment"). A hierarchy between the different kinds of services needs to be defined (e.g. "Cinema" and "Theatre" are "Entertainment"). In the EO example, we can define events of type hotel, dinner, walking, taking the bus, etc. Events provoke some changes into the user's state, e.g. changes in the location, the time, etc. We also attribute to each edge (event) a set of constraints representing *user preferences*.

Representation of constraints

As defined in (Rajpathak et al., 2001), we define a hierarchy between the constraints:

- Hard constraints cannot be violated under any circumstances;
- Soft constraints can be relaxed if necessary in order to reach the final schedule.

All these constraints are expressed using first-order logic expressions. A constraint will be expressed by a predicate. In the EO example (Figure 2), user state level constraints relate to the time and to the address: `(input-time <node> <value>)` and `(input-location <node> <value>)`.

The predicates should refer to a specific node in order to ease the detection of inconsistencies. A weight can be attributed to each constraint in order to express the degree of preference to the realisation of the condition, encoding the hard and soft constraints. Moreover, rules between these predicates can be defined according to the kind of service that is used. All the constraints will be used for checking the plan coherence and possibly resolve conflicts.

Resolution of the plan

This section details the second main part of the aggregation process, which corresponds to the resolution phase of the previous plan. This means that the non-instantiated plan has to be instantiated and verified according to the results provided by services.

From event to service types

In the third step, the aggregating entity should infer service types from events, by looking in the defined services hierarchy. For example, in the EO, we can use a mechanism that enables us to know that a restaurant event is the result of information given by a restaurant finder service, and that an entertainment could be the result of a film or play finder service.

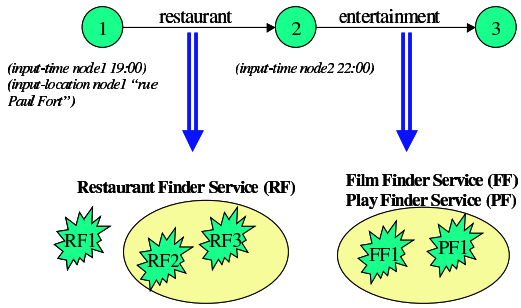

Figure 3: Selection of service providers

This mechanism can be a simple lookup table, or a more complex inferencing.

The fourth step will be divided into two sub-steps: Selection of a service provider and querying elementary service providers.

Selection of a service provider

Once the service type has been identified, providers of this service type, i.e. agents delivering this kind of service, should be selected among the possible candidates. The aggregating entity starts by searching in the DF (yellow pages service) the registered services. A pre-selection among different providers might be done thanks to some properties of the service description of agents registered within the DF (Figure 3).

If the aggregating agent finds no provider for a given service type, several possibilities may be considered: the plan can be rejected; or the plan can be revised: the user may be asked to review his choices, or the event for which there is no available service provider can even be removed from the original plan.

Querying elementary service providers

For each type of service that is selected, the *constraint processor* generates a query or a request that the service can understand, compliant with the constraints of the user (e.g. for the restaurant event, the constraints processor may generate a query including the type of cuisine and the facilities). This query is sent to the agent that returns the possible results (service instances such as a particular restaurant, a particular route, etc.) to the aggregating entity (Figure 4). Again, if no result is found at this sub-step for a given service type, the plan can be rejected or revised according to pre-defined rules.

Instantiated plan

This is the last step of the aggregation process. When getting back the results from the service agents, the aggregating entity will construct a set of graphs representing potential solutions. The coherence of these graphs has to be checked; only correct graphs

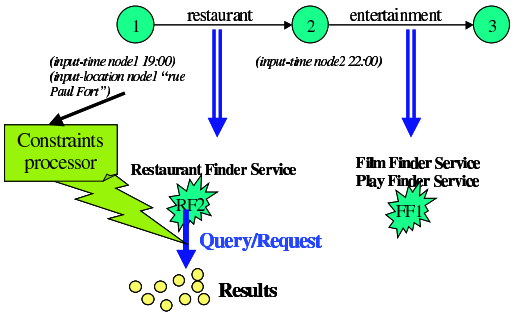

Figure 4: Retrieving results from service components

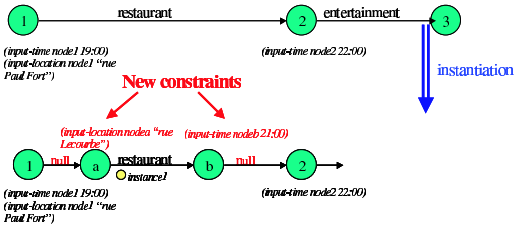

Figure 5: Instantiation of the plan

are kept. As mentioned in the aggregation process overview, according to the application domain and type we can use the right algorithm (from AI or OR domains) to return the first solution, all the possible solutions, or the optimal one(s). We call the obtained plans (if any) instantiated plans. A graph that may correspond to an instantiated plan is built from the abstract plan as follows:

- For each edge, two additional nodes are inserted with empty transitions (edges), in order to help in finding *conflicts* between planning constraints and in resolving these conflicts. Conflicts can arise from having different values for a given parameter (e.g. different locations for two successive nodes linked by an empty transition). These new nodes (Figure 5) may contain new constraints that are introduced because of the service instance results (e.g. a new input address, additional time constraints, etc.).
- If there is no conflict among the constraints between two nodes related by an empty edge, the two nodes are merged (see Figure 6, right part). Otherwise, we try to infer from this conflict the kind of event that could solve this issue, e.g. a transport can be added between a restaurant and a cinema located far from each other (see Figure 6, left part). If no event solving the issue is found, the plan should be rejected or revised. A rule-based engine, initiated by the designer, enriched from user interactions and by learning mechanisms, could perform the inference of the kind of event solving a conflict.

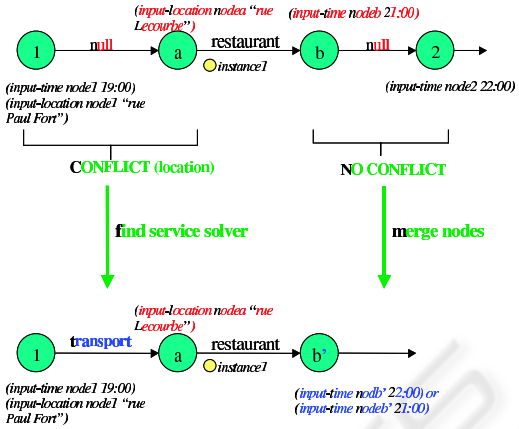

Figure 6: Detection and resolution of potential conflicts

Selection of a service instance

We should notice that the detection of conflicts is very difficult: constraints are represented by predicates and the potential conflicts can be detected using a theorem prover (like JTP (Frank, 1999)). The most difficult part of the aggregating entity resides in the level of intelligence the conflict solver is able to have. For example, in the EO case, if the user wants to go to a restaurant and then to a theatre, the evening organizer should not propose a transport if the two events are located in the same area.

4 EVENING ORGANIZER EXPERIENCE

This section presents a first application of our aggregation mechanism developed in the context of the Agentcities project.

A protoype of the aggregation architecture described above has been implemented, using the JADE (JADE, 1999; LEAP, 2001) agent platform. A JTP (Frank, 1999) engine provides some reasoning capabilities to our agents and a KIF support (Genesereth and Fikes, 1994) to exchange messages between them. This allows also agents to manipulate ontologies expressed in DAML+OIL (DAML+OIL, 2001), which have been written within the Agentcities project.

Our evening organizer agent interacts with a number of services (restaurants, route guidance, cinemas, hotels...) which allow to plan a complete evening. The interactions between the Evening Organizer and the services are quite simple, nonetheless we have implemented some basic negotiation schemes for selecting the most appropriate service. A simple Personal Agent acts as an interface between the user and the Evening Organizer. The evening plan is progressively

refined through interactions between the Evening Organizer and the Personal Agent, under the supervision of the user.

Currently, the taxonomy of services is poor, so the link between the event and the service type is static. We also used only basic reasoning capabilities to solve conflicts, but we will conceive and develop a more powerful reasoner for future versions. We tried some examples with this first prototype, and we obtained quite good results (like inferring and inserting a non-requested service to the proposed plan for user, e.g. a transport service for solving a location conflict).

5 CONCLUSION

In this paper, we have shown some steps towards a dynamic aggregation of agent-based services and a concrete achievement through the implementation of our Evening Organizer prototype.

During the development and experimental usage of the Evening Organizer architecture we have obtained valuable experience and a number of insights related to dynamic service aggregation. Nevertheless, our prototype shows that some issues remain. Thus, several areas requires further work:

- We currently rely on a very basic taxonomy of services for finding the relevant ones. We are investigating using DAML-S (DAML-S, 2002) for a more advanced service description, which would allow to have better reasoning capabilities;
- Improvements of the planning algorithms to choose the optimal or right aggregated services for the user, by keeping track of failed aggregation, for example, and reduce then their complexity;
- Improvements in the interactions between the Evening Organizer and the services.

While some of these improvements are only relevant to our implementation, others require some standardisation and agreement between researchers and developers involved in agent-based services design (services, user representatives, etc.)

ACKNOWLEDGEMENTS

The research described in this paper is partly supported by the EC project Agentcities.RTD (IST-2000-28385). The opinions expressed in this paper are those of the authors and are not necessarily those of the EU Agentcities.RTD partners.

REFERENCES

Agentcities (2003). http://www.agentcities.org.

Blum, A. L. and Furst, M. L. (1997). Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). Simple Object Access Protocol (SOAP) 1.1. http://www.w3.org/TR/SOAP.

DAML-S (2002). DAML-S Version 0.7. http://daml.org/services.

DAML+OIL (2001). DAML+OIL Specification. http://www.daml.org.

Frank, G. (1999). A General Interface for Interaction of Special-Purpose Reasoners within a Modular Reasoning System. In *Question Answering Systems, AAAI Symposium*, pages 57–62.

Genesereth, M. R. and Fikes, R. E. (1994). Kif version 3.0 reference manual. *Technical Report Logic Group Technical Report Logic-92-1*.

IBM (2001). Web Services Flow Language Version 1.0. Technical Report, http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf.

JADE (1999). Java Agent Development Framework. http://jade.tilab.com.

LEAP (2001). Lightweight Extensible Agent Platform. http://leap.crm-paris.com. IST-1999-10211.

Mc Ilraith, S., Son, T. C., and Zheng, H. (2001). Semantic Web Services. *IEEE Intelligent Systems*, 16(2):46–53.

Piccinelli, G. and Mokrushin, L. (2001). Dynamic Service Aggregation in Electronic Marketplaces. *HP Labs Technical Report, HPL-2001-31*.

Ponnekanti, S. R. and Fox, A. (2002). SWORD: A Developer Toolkit for Web Service Composition. In *WWW2002: The Eleventh International World Wide Web Conference*.

Rajpathak, S., E., M., and Roy, R. (2001). A Generic Task Ontology for Scheduling Applications. In *International Conference on Artificial Intelligence (IC-AI'2001)*, pages 1037–1043, Las Vegas, Nevada, USA.

UDDI (2001). Universal Description Discovery and Integration. http://www.uddi.org.

W3C (2001a). Semantic Web. http://www.w3.org/2001/sw.

W3C (2001b). Web Services Description Language Version 1.1. http://www.w3.org/TR/wsdl.

Weiss, G., e. (1999). *Multiagent systems: A Modern Approach to Distributed Artificial Intelligence*. The MIT Press, Cambridge, MA.

Wooldridge, M. and Jennings, N. R. (1999). The Cooperative Problem-Solving Process. *Journal of Logic and Computation*, 9(4):563–592.